ConSpec – a formal language for policy specification[1]

Irem Aktug[a], Katsiaryna Naliuka[b]

[a] *Royal Institute of Technology, Stockholm, Sweden*

[b] *University of Trento, Trento, Italy*

Abstract

The paper presents ConSpec, an automata based policy specification language. The language trades off clean semantics to language expressiveness; a formal semantics for the language is provided as security automata. ConSpec specifications can be used at different stages of the application lifecycle, rendering possible the formalization of various policy enforcement techniques.

1 Introduction

As mobile devices become increasingly popular, the problem of secure mobile application development gains importance. Mobile devices contain personal information, which users desire to protect. They also provide access to costly functionality, such as GSM services and GPRS connections. Hiding these resources from third-party applications would largely handicap application development for mobile platforms. It seems necessary to provide *controlled* access to the sensitive resources through fine-grained, at times application specific, constraints on execution.

A *security policy* selects a set of acceptable executions from all possible executions and thus can be used to define how and under what conditions a sensitive resource can be accessed. For instance, a user policy may limit the number of SMSs that are sent by an application per hour in order to prevent spamming. The decision for allowing access to a requested resource at a certain point of the program execution may depend on various factors, such as the previous actions of the application, the state of the environment, the parameters of the request etc. The user may want to forbid the sending of SMSs, for instance, after an application has accessed certain local files.

[1] Partially supported by S3MS project (http://s3ms.org).

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

A program adheres to a policy if all its executions are in the set of executions selected by the policy. Several techniques exist to ensure that an application complies to a policy. *Static verification* techniques, such as model checking, analyze the program code in order to construct a mathematical proof that no execution of the program can violate the policy. Such an analysis is thorough and provides full assurance, at the same time, it is costly and often requires human interaction. *Runtime monitoring* can be used as an alternative to static checking. This security enforcement mechanism observes the behavior of a target program and terminates it if it does not respect the policy. Monitoring can effectively enforce many interesting security properties [14]. However, it creates performance overhead since each *security relevant action* of the program should be detected and checked against the policy. Monitoring may be performed *explicitly*, i.e. by a separate program which is *co-executed* (executed in parallel) with the untrusted application. Due to expensive interprocess communication however, this technique is costly. In order to reduce this overhead, the monitor can instead be *inlined* in the untrusted program [5]. Then, the code of the program is interleaved with the code of the monitor.

We describe here first how security specifications can be enforced at the three stages of the application lifecycle: the *development*, *installation* and *runtime* phases. The approach that we present here combines static verification and monitoring to enforce security properties on mobile devices in the most effective way. We associate with the application a *contract* [4], a piece of data that describes its *security-relevant behavior* which simplifies tasks related to security enforcement. A framework which spans different stages of the application lifecycle and combines different techniques for ensuring compliance benefits from a common language for policy specification. In turn, the different aspects of the framework imposes different restrictions on such a language. The main contribution of this paper is the language *ConSpec* (Contract Specification Language) which can be used for specifying both user policies and application contracts. A semantics for ConSpec is provided and the formal treatment of several activities in the framework is briefly explained based on this semantics.

The paper is structured as follows. In §2, we describe the lifecycle of the application paired with its contract. In §3, we discuss design decisions behind ConSpec, present its syntax and give a formal semantics to the language. Discussion of the related work and final remarks §5 end the paper.

2 Security Enforcement in the Application Lifecycle

In this section, we describe how security enforcement techniques can be applied throughout the lifecycle of an application and how the goals of all participants can be achieved in the contract-aware framework. The lifecycle of the application and the activities associated with each development phase are illustrated in Fig. 1. We make use of the following scenario in the rest of the section:

Companies Alpha and Beta produce applications for mobile devices. Alpha develops application `Weather` *that every morning at a user-defined time sends an SMS message to the operator's weather service and displays to the user the forecast it receives. Application* `HappyBirthday`*, produced by Beta, checks the user's address*

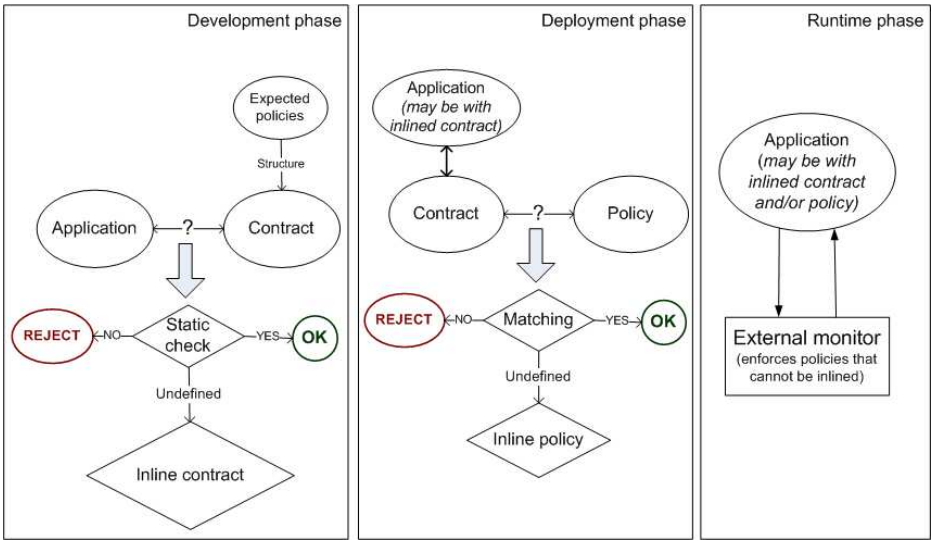

Fig. 1. Security enforcement through application development phases

book and sends a congratulation SMS to each contact that has a birthday. Both companies want their applications to be used by as many users as possible.

Alice is a user of the mobile device. She wants to download and use the third-party applications. But she does not want these applications to break the policy "An application must not send more than 5 SMSs messages per day".

Development phase We assume that the developer is aware of typical security policies and is willing to keep his application in conformance with them. From the policies he learns which actions of the application are security-relevant. Using this information he provides the application with the contract, which specifies the intended security-relevant behavior of application. At this phase, the policy language is used for expressing this contract. The compliance of the contract and the application can be checked, for instance, using static verification by a trusted third party, who then signs the application and the contract by its private key. This analysis is performed by powerful machines rather than the mobile devices, and can make use of knowledge available to the developer (e.g. program specifications, annotations derived from the source code etc). Instead of signing the application with a private key, *proof-carrying code* method [13] can be used to convey assurance in program-contract compliance. The application and the contract are supplied with an easy-to-check proof of their compliance. If contract compliance can not be statically verified, then an execution monitor can be inlined in the program at this stage so that the compliance is ensured at runtime.

In our example scenario Alpha and Beta are not aware about the particular limit of SMS messages that Alice allows. But they know that the number of messages matters. Therefore, Alpha supplies the `Weather` *application with the contract that the application sends only one message per day. However, Beta developers cannot tell in advance how many messages their application sends per day. For this reason, the contract for their application is more complex. It tells that the application will send one message to every contact from the address book that has a birthday.*

Installation phase Before the program is installed on the device, a formal

check is needed to show that the security-relevant behavior of the application given by the contract is acceptable by the user policy. If policies and contracts are captured with automata on infinite strings, the problem of matching a policy against a contract reduces to the language containment problem for such automata. The complexity of this task (for example, the problem of language containment is undecidable for two context-free languages [9,10]) severely restricts the expressive power of the policy language. When the problem is decidable, however, contract-policy matching is much simpler than checking the program itself, and is more likely to be feasible on a mobile device [4].

In our scenario, the contract of `Weather` *can be matched against Alice's policy "No more than 5 messages per day". So this application is permitted to run at the device without any modifications. But the contract of* `HappyBirthday` *cannot be matched due to the lack of the precise number of messages the program can send. It can still run on the device, after it undergoes inlining.*

A policy that is not covered by the contract can be enforced by monitoring. If the program is to be monitored explicitly, hooks that notify the monitor about ongoing security-relevant activity should be injected to the program in the installation stage, i.e. prior to execution. If the monitoring task is to be optimized in order to reduce runtime overhead, the monitor for the desired policy should be inlined into the program at this stage.

For instance, application `Weather` *does not need to undergo inlining since its compliance to the user's policy has already been verified. But a monitor for Alice's policy is inlined in the* `HappyBirthday` *application to ensure that the application does not send more than 5 messages.*

Runtime At runtime, the behavior of an application may be checked against a policy by monitoring. Because of the performance overhead created by monitoring, it is preferable to use static methods described above and leave as little work to runtime as possible. But in many cases, the application of other techniques is not feasible (or not even possible due to, for example, the unavailability of the source code), and runtime monitoring is the only solution to protect a system.

In our example, application `Weather` *will not be monitored. But* `HappyBirthday` *will, and if a violation is detected (that is, if the application is trying to send the 6th message), it will be terminated. The behavior of the program will otherwise be unaltered (except for the slight performance deterioration due to monitoring) and the user will be able to freely enjoy its functionality.*

3 ConSpec Language

The intention behind ConSpec is to design a language that can be exploited both for specification of requirements and for the description of the security-relevant behavior of actual systems. For this reason, the formalism selected is based on automata, which have been used for both purposes. For instance, the SPIN tool [8] inputs system specifications as models written in the guarded-command language Promela and performs model checking on the Büchi automata extracted from these models. Security properties are also expressed as automata in various approaches (e.g. [14,15]).

ConSpec is strongly inspired by the policy specification language PSLang, which was developed by Erlingsson and Schneider [5] for runtime monitoring. PSLang policies consist of a set of variable declarations, followed by a list of security relevant events, where each event is accompanied by a piece of Java-like code that specifies how the security state variables should be updated in case the event is encountered in the current state. PSLang policies make monitor inlining simple: the updates provided by the user can be almost directly inserted into the target program. However, this leads for making specifications less formal. A policy text is intended to encode a security automaton: the state variables represent the automaton states and updates represent transitions. While this intuition is given, the exact way to extract the automaton from a PSLang policy is not provided. Such a task is not trivial due to the power of the programming language constructs that can be used in the updates.

Further we provide a formal semantics which maps ConSpec policies to formal objects that can be used in constructing mathematical proofs. It is important to note that ConSpec is a more restricted language than PSLang; this is a design decision taken in order to allow application of formal methods for all stages of the development process, and not just runtime monitoring. More specifically, ConSpec does not allow arbitrary types in representing the security state and restricts the way the security state variables are updated. We have used a guarded-command language for the updates where the guards are side-effect free and commands do not contain loops. The simplicity of the language then allows for a comparatively simple semantics. While the general ConSpec, which is the common language for all tasks in the application lifecycle, is to be kept as simple as possible, specific tasks may allow certain extensions. For instance, while putting conditions on heap objects make matching undecidable, these are easily handled by monitor inlining. A table that shows which features can be supported by different tasks is included in the Appendix.

ConSpec has a construct (`Scope`) for expressing security requirements on different levels. Case studies [16] show that many interesting real-life policies concern the entire execution history rather than a single run of the application, although most policy languages (including PSLang) do not contain the feature of distinguishing between events in the current run and in the previous runs. ConSpec is expressive enough to write policies on multiple executions of the same application (scope `Multisession`) and on executions of all applications of a system (scope `Global`), in addition to policies on a single execution of the application (scope `Session`) and on lifetimes of objects of a certain class (scope `Object`).

ConSpec Syntax Figure 2 summarizes the syntax of ConSpec policies. Before the actual policy, ConSpec files set a limit on values of the type `int` which consist of some initial segment of natural numbers. Similarly, maximum length of strings are specified. We have skipped these in the example policy. Persistent state declaration that follows the multi-session and global scopes is similar to security state declaration and aims to specify the state that is preserved across single executions.

An event clause (see Figure 2(b) for syntax) gives us a security relevant action and its modifier. The events that we are considering security relevant are method invocations. These methods can be system calls or methods provided by an API.

```
MAXINT M
MAXLEN N
SCOPE <Object ClassName
      | Session
      | Multisession PersistentStateDec
      | Global PersistentStateDec>

SECURITY STATE
    PrimType SecVar1 = InitVal1
             ⋮
    PrimType SecVarN = InitValN

Clause1
             ⋮
ClauseK
```

(a) Policy Syntax

```
<BEFORE
 |EXCEPTIONAL
 |AFTER [Type Name = ]> Signature

   PERFORM
   Guard1          -> {UpdateBlock1}
          ⋮
   GuardM          -> {UpdateBlockM}
   [ ELSE          -> {UpdateBlock} ]
```

(b) Event Clause Syntax

Fig. 2. ConSpec Syntax

In order to resolve which method is of interest in case of overloading, the argument types of the method is to be specified as part of the action specification. The security relevant action is then fully specified by its *signature* which consists of the name of the method, the class to which the method belongs and the types of its arguments. The signature of an event clause is defined as the signature of the method associated with it. In ConSpec policies, all event clauses with the same modifier have a unique signature. This restriction means that one can not, for example, have two `BEFORE` clauses for the same method. The restriction has been imposed in order to ensure determinism. Notice that since the signature does not include the type of the return variable, it is not possible to have two `AFTER` event clauses for the same method, even if they do not agree on the return variable types. The modifier states when the update to the state will be performed: will the guards be evaluated before the event, after the event or immediately after the throwing of an exception by the event.

The event specification is followed by a sequence of guard-update block pairs. The update specifies how a state will be updated for the security relevant action while the guard selects the states, which the particular update will apply, as a subset of all states. The guards are considered from top to bottom. In case none of them is true, there is no transition for that action from the current state. If an `ELSE` block is present, however, the update of this block is executed in case none of the guards above it are satisfied. The guard is a side-effect free boolean expression which can mention only the set of argument values (and the return value for the case of `AFTER` modifier), and the security state. The update block begins with declarations of the local variables, which have the current block as their scope. A list of assignments to local variables and security state variables follow the declarations. If no assignments are present, the update block consists of the statement `skip`.

The expression language of ConSpec has been designed to ensure that checking language containment of the induced automata (the matching problem) is decidable. The security state variables of ConSpec are restricted to the primitive types (*`PrimType`*): booleans, integers, and strings. All other variables can be of the general type (*`Type`*), which includes both primitive types and classes. The boolean expressions in guards can also include field accesses using object references . Field access is expressed by the "`.`" operator. The expressions on integers are built using basic arithmetic and comparison operators. Strings can be checked for equality and

the prefix relation using the functions `equals` and `beginsWith` respectively.

An example policy The policy "*An application must not overwrite local files and after it has accessed an existing file, it should get approval from the user before opening a connection*" is expressed in ConSpec as follows:

```
SCOPE Session SECURITY STATE
      bool accessed= false;
      bool permission = false;

BEFORE File.Open(string path, string mode, string access) PERFORM
      mode.equals("CreateNew") -> { skip; }
      mode.equals("Open") && access.equals("OpenRead")  -> { accessed= true; }


BEFORE Connection.Open(string type, string address) PERFORM
      !accessed -> { permission= false; }
      accessed && permission -> { permission = false; }

AFTER string answer= GUI.AskConnect() PERFORM
      answer.equals("Yes") -> { permission=true; }
```

We begin by specifying that the policy applies to each single execution of an application. Scope declaration is followed by the *security state declaration*: the security state of the example policy is represented by the boolean variables `accessed` and `permission`, which are both false initially to mark, respectively, that no file has been accessed and that no permissions are granted when the program begins executing. The example policy contains three *event clauses* that state the conditions for and effect of the security relevant actions: call to the method `File.Open`, call to the method `Connection.Open` and return from the method `GUI.AskConnect`. The types of the method arguments are specified along with representative names, which have the event clause as their scope. The *modifiers* BEFORE and AFTER mark whether the call of or the normal return from the method specified in the event clause is security relevant (exceptional returns can be specified by the modifier EXCEPTIONAL). Event clauses contain guards and associated updates to the security state variables.

ConSpec Semantics We give semantics to policies written in ConSpec through a particular class of security automata which we term *ConSpec automata*.

Notation. In the text below, we fix a set of class names $\mathbb{C}$, and method names $\mathbb{M}$ ranged over by $\mathbf{c} \in \mathbb{C}$ and $\mathbf{m} \in \mathbb{M}$, respectively. We assume that types are ranged over by τ. The set of all values of type τ is denoted as $\|\tau\|$. The set of all values is $Val = \bigcup_{\tau} \|\tau\|$ while the set of values of the type `int`, `boolean` or `string` is $PrimVal$. We take τ_{LOC} to be the object reference type, and $LOC = \|\tau_{LOC}\|$ is the set of addresses in the heap. Heaps map locations to functions which in turn map a set of field names to values. Heaps are then defined as partial functions from addresses (from the set LOC) to objects. We take Θ as the set of all possible heaps, so for $h \in \Theta$, $h : LOC \rightharpoonup FVar \rightarrow Val$.

ConSpec Automata In ConSpec automata, security relevant actions are method calls, represented by the class name and the method name of the method, along with a sequence of values that represent the actual argument list of the method. We partition the set of security relevant actions into a set of *before actions* $A^{\flat}$ and a set of *after actions* $A^{\sharp}$, corresponding to method invocations and returns. Both refer to the heap prior to method invocation, while the latter also refers to the heap upon termination and to a return value from $RVal = Val \cup \{\bot, \varepsilon\}$ where ε and $\bot$ are used to model return from a void method and return on an exception

raised during the method call.

$$A^\flat \subseteq \mathbb{C} \times \mathbb{M} \times \mathit{Val}^* \times \Theta$$
$$A^\sharp \subseteq \mathit{RVal} \times \mathbb{C} \times \mathbb{M} \times \mathit{Val}^* \times \Theta \times \Theta$$

The partitioning on security relevant actions induces a corresponding partitioning on the transition function δ of ConSpec automata. We present a deterministic version of security automata.

Definition 3.1 (ConSpec Automaton) *A* ConSpec automaton *is a tuple* $\mathcal{A} = (Q, A, \delta, q_0)$*, where:*

(i) Q *is a countable set of states,*

(ii) $q_0 \in Q$ *is the initial state,*

(iii) $A = A^\flat \cup A^\sharp$ *is a countable set of security relevant actions as described above, and*

(iv) $\delta = \delta^\flat \cup \delta^\sharp$ *is a (partial) transition function, where* $\delta^\flat : Q \times A^\flat \rightharpoonup Q$ *and* $\delta^\sharp : Q \times A^\sharp \rightharpoonup Q$*.*

The *enforcement language* of a ConSpec automaton $\mathcal{A}$ is defined as the set $L_\mathcal{A} \cup L_\mathcal{A} \cdot A^\sharp$, where $L_\mathcal{A}$ is the language of $\mathcal{A}$ in the standard sense. It is the enforcement language which defines the security policy induced by a ConSpec automaton.

Automaton Extraction From Policy Text The semantics of a ConSpec policy $\mathcal{P}$ is given in terms of a ConSpec automaton $\mathcal{A}_\mathcal{P} = (Q, A, \delta, q_0)$ as described below.

States. The set of states Q of $\mathcal{A}_\mathcal{P}$, also called security states, is determined by the declarations in the `SECURITY STATE` block of the policy $\mathcal{P}$. Consider the security state declaration of $\mathcal{P}$:

$$\texttt{SECURITY STATE}\ \begin{array}{l}\tau_{s_1}\ s_1 = v_1\\ \vdots\\ \tau_{s_k}\ s_k = v_k\end{array}$$

The set of variable names that are induced by such a state declaration is the set of *security state variables.* $\mathit{SVar} = \{s_1, \ldots, s_k\}$. The states $q \in Q$ of the automaton are mappings from variable names to values which respect the types of the security state variables: $q : \mathit{SVar} \to \mathit{Val}$. The initial state q_0 simply maps the security state variables to their initial values: $\forall s_i \in \mathit{SVar}.\ q_0(s_i) = v_i$.

Actions. The actions A of the automaton are determined by the events mentioned in event clauses of the policy. An action $a = \langle \mathbf{c}, \mathbf{m}, (v_1, \ldots, v_n), h\rangle$ is a security relevant before action, $a \in A^\flat$, if and only if the ConSpec policy contains an event clause:

`BEFORE` $\mathbf{c}.\mathbf{m}\,(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n)$ `<body>`

where $v_1 \in \|\tau_1\|, \ldots, v_n \in \|\tau_n\|$. Similarly, an action $a = \langle \mathbf{v}, \mathbf{c}, \mathbf{m}, (v_1, \ldots, v_n), h, h'\rangle$ where $\mathbf{v}$ is a value v, ε, or $\bot$, is a security relevant after action, $a \in A^\sharp$, if and only if the ConSpec policy contains an event clause:

`AFTER` $\tau\ x = \mathbf{c}.\mathbf{m}\,(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n)$ `<body>` or

`AFTER` $\mathbf{c}.\mathbf{m}\,(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n)$ `<body>` or

`EXCEPTIONAL` $\mathbf{c}.\mathbf{m}\,(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n)$ `<body>`

respectively, where $v_1 \in \|\tau_1\|, \ldots, v_n \in \|\tau_n\|$ and $v \in \|\tau\|$.

Transitions. Each event clause of the policy induces a partial transition function. The transition functions $\delta^\flat$ and $\delta^\sharp$ of the automaton are the union of the partial

functions corresponding to event clauses with the `BEFORE` and `AFTER`/`EXCEPTIONAL` modifier, respectively. The definition of the partial functions is similar for both types of event clauses; the difference is that the transitions of $\delta^\sharp$ also contain the return value when the method has some other return type than `void` (and the constant ε when the return type is `void`). For brevity, here we only describe this more general case.

Consider an `AFTER` event clause $\varphi^\sharp$:

`AFTER` $\tau\ x = \mathbf{c}.\mathbf{m}\,(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n)$
`PERFORM`
$Guard_1$ `->` $UpdateBlock_1$
$\vdots$
$Guard_m$ `->` $UpdateBlock_m$

Let $AVar = \{x_1, \ldots, x_n\}$ be the set of formal arguments of the event and $PVar = \{x\} \cup AVar$ be the set of all program variables of the event clause. Below, let states $q \in Q$ be as defined above, and let $\sigma : PVar \to Val$ range over the set Σ of mappings from program variables to values which respect the declared types of the variables. For guards $Guard_j$, or G_j for short, and every blocks $UpdateBlock_j$, or U_j for short, of $\varphi^\sharp$, we assume the semantic functions:

$$[\![G_j]\!] : Q \times \Sigma \times \Theta \times \Theta \to \{True, False\}$$
$$[\![U_j]\!] : Q \times \Sigma \times \Theta \times \Theta \to Q$$

where the two heaps in the function types refer to the heap of the program before and after the execution of the method call, respectively. The `ELSE` keyword used as a guard would then correspond to the guard `true`.

The ConSpec language refers to fields of object references by using the standard "`.`" notation. Semantics of such expressions are relativized on heaps. The heap is not changed by the automata, but is only used to look up fields of object references. Below, we denote the heap before the call with $h^\flat$, and the heap after the call with $h^\sharp$. Then, the value of field access expressions are as follows:

$$Var.Field = \begin{cases} h^\flat(\sigma(Var))(Field) & \text{if } Var \in AVar \\ h^\sharp(\sigma(Var))(Field) & \text{if } Var = x \end{cases}$$

Then, every event clause $\varphi^\sharp$ induces a partial mapping $p^\sharp : Q \times A^\sharp \rightharpoonup Q$ as follows. For a security state q and after action $a = \langle \mathbf{v}, \mathbf{c}, \mathbf{m}, (v_1, \ldots, v_n), h, h' \rangle$, we define $p^\sharp(q, a) = q'$ if there exists $1 \leq i \leq m$ such that:

- $[\![G_j]\!](q, \sigma, h^\flat, h^\sharp)$ and
- $\forall i < j.\neg([\![G_i]\!](q, \sigma, h^\flat, h^\sharp))$ and
- $[\![U_j]\!](q, \sigma, h^\flat, h^\sharp) = q'$

where $\sigma : PVar \to Val$ is defined by the correspondence of actual to formal parameters induced by $(v_1, \ldots, v_n)$ and the return value $\mathbf{v}$. This definition captures that the guards are evaluated in order from top to bottom in order to select the right update block.

Finally, the *after*-transition function $\delta^\sharp$ is the union of the functions induced by each event clause (with disjoint domains):

$$\delta^\sharp = \biguplus_{\varphi^\sharp \in \mathcal{P}} p^\sharp$$

4 ConSpec in Use

The main advantage of ConSpec is that it allows for a formal treatment of the various enforcement techniques mentioned in Section 2 through its automata-based semantics. Here we briefly explain how this can be achieved.

Static check If the contract of the application is formalized in ConSpec model then static check can be performed by translating the corresponding security automaton into Spec# constraints and verifying the resulting specification. The approach is described in more details in [1].

Matching One way to match a ConSpec contract against a ConSpec policy is to check that the language of the contract automaton is included in the language of the policy automaton. Since the domains of the security state variables are bounded, the extracted automata have finitely many states and standard methods for checking language inclusion for automata can be facilitated for contract-policy matching (see for instance [3]).

Monitoring Given a program and a ConSpec policy with scope `Session`, the concept of monitoring can be formalized by defining the co-execution of the corresponding ConSpec automaton with the program. Such co-executions are a subset of the set of interleavings of the individual executions of the program and the automaton. Co-executions satisfy the following condition: when the execution of the program component is projected to its security relevant action executions, each *before* action is immediately preceded by a transition of the automaton for the same action; dually, each *after* action is immediately followed by a corresponding automaton transition. Therefore it is simple to show that the program component of the co-execution adheres to the given policy, as the co-execution includes an accepting trace of the automaton for the program execution.

Monitor Inlining Inlining a ConSpec policy with scope `Session` can be performed similar to inlining a PSLang policy (see [5] for details). A class definition is added to the target program which stores the security state variables. Then the program is rewritten so that each security relevant method call is wrapped with code compiled from the corresponding event clause(s) of the policy. Such a code segment evaluates the guards of the event clause from top to bottom and executes the updates associated with the first guard that is satisfied. If none of the guards evaluate to true, the program is terminated. The modifier of the event clause determines where this segment is placed relative to the method call. The correctness of such a monitor inlining scheme can be proven by setting up a bisimulation relation between the states of the inlined program and the states of the co-execution of the original program with the ConSpec automaton (of the policy).

5 Related work and conclusion

There exists a number of automata-based languages for security policy specification. Amongst these, ConSpec is closest to PSLang [5] which has also introduced the modifiers used in ConSpec. The language is intended solely for runtime monitoring and freely uses programming language constructs such as abstractions and functions.

This enables a larger class of policies to be specified but also complicates the task of providing a formal semantics. Since the authors do not provide such a formalisation, their monitor inlining algorithm for PSLang is to be trusted on intuition as no proof of its correctness can be constructed.

The Polymer language [2] has the same drawback. Polymer policies consist of Java classes which, when inlined, may trigger various action in case of violation. For instance it is possible to execute some recovery action as a response to the violation, after which the application is allowed to progress. Polymer policies implement *edit automata* [12], which extend security automata([6]). But the correctness of the Polymer policy inlining cannot be proven either, as its semantics is not formally presented.

Many languages use logic-based formalisms to express security properties [7]. However, it seems that these languages are less convenient for specification of the existing systems automata-based languages, as it is hard to represent the full behavior of the system through a limited set of temporal-logic properties. Yet, in our framework we need a formalism convenient for specification of both programs and requirements to them. Moreover, temporal logic formulae can be translated to automata by applying a tableaux procedure [11].

Model-carrying code (MCC) [15] is based on the idea of supplying untrusted code with additional information to simplify its verification against user policies. In MCC, this additional information is an extended FSA (EFSA) that represents the model of the program. This approach has much in common with ours, and EFSA is much similar to our ConSpec automata. However, for full EFSA verification algorithms are not developed. Therefore, the current framework for MCC allows only equality/disequality conditions of the variables, while our language allows more sophisticated expressions, including basic arithmetic operations and comparisons of numeric values. Also, our framework does not rely entirely on monitoring for enforcing code-contract compliance. In many cases the compliance can be verified statically and run without performance overhead.

In this paper we presented the policy language ConSpec, which has been designed for formalizing security requirements as well as representing the security-relevant behavior of the application. ConSpec specifications can be used for various tasks during all stages of the application lifecycle to ensure that the application conforms to the user policy. The main features of ConSpec are this universality and its tight connection with the underlying formalism, which is a fundamental component of formal proofs of policy adherence.

In the scope of the S3MS project, we are formalizing enforcement techniques using ConSpec as summarized in Section 4. However, we currently consider sequential programs only. As future work, we aim to extend our approach to applications where multiple threads can perform security-relevant actions. Such a setting brings about synchronization issues as mutually dependent events may occur in different threads and data used by the monitor for decision-making may be shared between threads. Thus, formalizing monitoring of multi-threading environments emerges as a challenging problem.

6 Acknowledgements

The authors thank Dilian Gurov and Fabio Massacci for valuable comments and discussions.

References

[1] Aktug, I., D. Gurov, F. Piessens, F. Seehusen, D. Vanoverberghe and E. Vétillard, *Static analysis algorithms and tools for code-contract compliance*, Public Deliverable D3.1.2, S3MS, http://s3ms.org (2006).

[2] Bauer, L., J. Ligatti and D. Walker, *A language and system for composing security policies*, Technical Report TR-681-03, Princeton University (2004).

[3] Courcoubetis, C., M. Vardi, P. Wolper and M. Yannakakis, *Memory-efficient algorithms for the verification of temporal properties*, Form. Methods Syst. Des. **1** (1992), pp. 275–288.

[4] Dragoni, N., F. Massacci, K. Naliuka and I. Siahaan, *Security-by-contract: Toward a semantics for digital signatures on mobile code*, in: *European PKI Workshop: Theory and Practice (to appear)*, 2007.

[5] Erlingsson, U., "The inlined reference monitor approach to security policy enforcement," Ph.D. thesis, Dep. of Computer Science, Cornell University (2004).

[6] Hamlen, K. W., G. Morrisett and F. B. Schneider, *Computability classes for enforcement mechanisms*, ACM Trans. Prog. Lang. Syst. **28** (2006), pp. 175–205.

[7] Havelund, K. and G. Rosu, *Efficient monitoring of safety properties*, Int. J. on Software Tools for Technology Transfer (2004).

[8] Holzmann, G. J., *The model checker SPIN*, Software Engineering **23** (1997), pp. 279–295.

[9] Hopcroft, J. E., *On the equivalence and containment problems for context-free languages*, Theory of Computing Systems **3** (1969), pp. 119–124.

[10] Hunt, H. B. and D. J. Rosenkrantz, *On equivalence and containment problems for formal languages*, J. ACM **24** (1977), pp. 387–396.

[11] Kesten, Y., Z. Manna, H. McGuire and A. Pnueli, *A decision algorithm for full propositional temporal logic*, in: *CAV*, 1993, pp. 97–109.

[12] Ligatti, J., L. Bauer and D. Walker, *Edit automata: enforcement mechanisms for run-time security policies*, Int. J. of Information Security (2003).

[13] Necula, G. C., *Proof-carrying code*, in: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[14] Schneider, F. B., *Enforceable security policies*, ACM Transactions on Information and System Security **3** (2000), pp. 30–50.

[15] Sekar, R., V. Venkatakrishnan, S. Basu, S. Bhatkar and D. C. DuVarney, *Model-carrying code: a practical approach for safe execution of untrusted applications*, ACM SIGOPS Operating Systems Review (2003).

[16] Zobel, A., C. Simoni, D. Piazza, X. Nuez and D. Rodriguez, *Business case and security requirements*, Public Deliverable D5.1.1, S3MS, http://s3ms.org (2006).

Appendix: ConSpec Features for Different Tasks

As extensions to the current language ConSpec, introducing object reference and list types as security state types emerge as beneficial features considering real-life policies. The list type makes simple iteration meaningful to include in the update language, to enable, for instance, updating all elements of a list. The update language then can be extended with a simple construct that iterates over flat lists. Extensions to the language should be considered thoroughly, as these may introduce undecidability of various tasks identified in our framework. Here we provide a table that shows which extensions to the language can be handled by the various tasks in the framework. The constructs of ConSpec are specified in the rows of the tables below, whereas the activities are specified in the columns.

Construct	**Static analysis**	**Monitoring**	**Matching**
Policy scope			
Scope object	+	+	+
Scope session	+	+	+
Scope multi session	-	+	+
Scope global	-	+	+
State declaration			
Bounded integers	-	+	+
Bounded strings	-	+	+
Booleans	+	+	+
Object ref.	-	+	-
Lists (of the above)	-	+	-
Unbounded versions the above types	-	+	-
Command			
Local variable declaration	-	+	-
Assignment	+	+	+
Conditional branch	-	+	-
For-loop	-	+	-