Gate automata-driven run-time enforcement[☆,☆☆]

Gabriele Costa[a,b], Ilaria Matteucci[b]

[a] *Università di Pisa*
[b] *Istituto di Informatica e Telematica-CNR*
name.surname@iit.cnr.it

Abstract

Security and trust represent two different perspectives on the problem of guaranteeing the correct interaction among software components.

Gate automata have been proposed as a formalism for the specification of both security and trust policies in the scope of the Security-by-Contract-with-Trust (S×C×T) framework. Indeed, they watch the execution of a target program, possibly modifying its behaviour, and produce a feedback for the trust management system. The level of trust changes the environment settings by dynamically activating/deactivating some of the defined gate automata.

The goal of this paper is to present gate automata and to show a gate automata-driven strategy for the run-time enforcement in the S×C×T.

Keywords: Security-by-Contract-with-Trust, gate automata, interface automata, contract monitoring, run-time enforcement.
2010 MSC: 68Q60, 68Q45

1. Introduction

In the last decades, the number of devices, used in our daily life has been rapidly growing. Also, the computational capabilities of such devices have increased over and over. Beyond the clear advantages for their users, *e.g.*, in terms of reachability and connectivity, this trend also exposed to new vulnerabilities and security threats. Often, the users try to mitigate these risks using some informal, trust-based evaluation for avoiding interactions with potentially malicious agents.

Here we mainly focus on the Security-by-Contract-with-Trust (S×C×T) [6], *i.e.*, a *contract-based* approach able to manage the trust levels of the applications

[☆] This paper is an extended version of the paper [8] presented at IMIS 2011.

[☆☆] Work partially supported by EU-funded project FP7-231167 CONNECT, by EU-funded project FP7-257930 ANIKETOS and EU-funded project FP7-256980 NESSOS.

and guarantee the security requirements at run-time.

In this paper we show an implementation of the S×C×T framework run-time support using *gate automata*. We advocate gate automata as a unique formalism for dealing with security and trust [8]. In particular, we show how they can be used for encoding both applications' contracts and security policies in the S×C×T implementation. Roughly, after downloading an application from a remote provider, according to its identity, we associate the code to a certain trust domain. The lower is the level of trust the stronger, *i.e.*, the more restrictive, is the security policy we enforce on the application. While executing the application, the run-time enforcement support controls its behaviour. If a violation occurs, *i.e.*, the application tries to perform an action that is not allowed by the current security configuration, the enforcement system reacts, possibly decreasing the level of trust of the application. Trust updating may lead to a reorganisation of the run-time enforcement mechanism that may enforce a different, *e.g.*, more restrictive, security policy.

This paper is organized as follows: Section 2 presents gate automata definition and the ConSpec contract policy language by showing how the semantics of latter can be given through the former. Section 3 shows the structure of a S×C×T implementation using gate automata. Section 4 compares our work with the other in the literature and Section 5 concludes the paper with some considerations about the future research directions in this field.

2. Gate Automata

In this section we formally introduce *gate automata* and their properties. We also show the *ConSpec* [1] language by recalling its syntax and by giving its semantics in terms of gate automata.

Definition 2.1. *A gate automaton $\mathcal{G}$ is a 4-tuple $\langle V, \imath, A, T\rangle$ where: V is a finite set of states; $\imath \in V$ is the initial state; A is a set of actions (being $\bar{A}$ the set of the complementary actions of A); $T \subseteq V \times (A \cup \bar{A} \cup \{\blacktriangle, \blacktriangledown\}) \times V$ is a set of labelled transitions such that:*

1. $(v, a, u) \in T \wedge (v, b, w) \in T \wedge a = b \iff u = w$
2. $\forall (v, a, u) \in T . a \in \bar{A} \cup \{\blacktriangle, \blacktriangledown\} \implies \nexists\, b, w . b \neq a \wedge (v, b, w) \in T$

A gate automaton processes a sequence of actions possibly modifying it. The transitions of the automaton can be labelled with input (*i.e.*, $\alpha \in A$) or output (*i.e.*, $\bar{\alpha} \in \bar{A}$) actions. An input action is generated by some actions source, *e.g.*, a running program, while output actions are fired by the automaton itself. Gate automata can also perform two special operations, ▲ and ▼, which, respectively, increase and decrease the trust weight corresponding to the source of the actions. Where it improves the readability, we use $v \xrightarrow{\alpha} w$ in place of $(v, \alpha, w) \in T$ and $v \not\xrightarrow{\alpha}$ for $\nexists w\,.\,(v, \alpha, w) \in T$.

2.1. Gate automata and interface automata

A gate automaton can be instantiated to a corresponding interface automaton [2]. Hence, we use interface automata for giving an operational semantics to the security policies defined through our gate automata.

Definition 2.2. *An instantiation of a gate automaton $\mathcal{G}$ over a index k, denoted by $\mathbf{G}_k$, is an interface automaton $P = \langle V_P, \{\iota\}, A_k^{\mathcal{I}}, A_{k+1}^{\mathcal{O}}, \{\blacktriangle, \blacktriangledown\}, T_P\rangle$ where:*

- *$V_P = V \cup V_{id}$ is the finite set of states (where $V_{id} = \{v_{id}^{\alpha} : v \in V \ \wedge \ v \overset{\alpha}{\not\mapsto} \wedge \forall \beta \in \bar{A} \cup \{\blacktriangle, \blacktriangledown\}. v \overset{\beta}{\not\mapsto}\}$)*
- *$A_k^{\mathcal{I}} = \{\langle \alpha, k\rangle : \alpha \in A\}$ is the input alphabet;*
- *$A_{k+1}^{\mathcal{O}} = \{\langle \alpha, k+1\rangle : \alpha \in A\}$ is the output alphabet;*
- *T_P is a set of transitions defined as:*

$$\begin{array}{rcl} T_P & = & \{(v, \langle \alpha, k\rangle, w) : (v, \alpha, w) \in T\} \cup \{(v, \langle \alpha, k+1\rangle, w) : (v, \bar{\alpha}, w) \in T\} \\ & \cup & \{(v, \blacklozenge, w) : (v, \blacklozenge, w) \in T\} \cup \{(v, \langle \alpha, k\rangle, v_{id}^{\alpha}) : v_{id}^{\alpha} \in V_{id}\} \\ & \cup & \{(v_{id}^{\alpha}, \langle \alpha, k+1\rangle, v) : v_{id}^{\alpha} \in V_{id}\} \end{array}$$

where $\blacklozenge \in \{\blacktriangle, \blacktriangledown\}$

The semantics of an instantiation $\mathbf{G}_k$ of a gate automaton $\mathcal{G}$ is defined in terms of *reaction sequences*. Intuitively, a reaction sequence is a trace of output and internal actions fired by an interface automaton after reading one input symbol. We start by extending the definition of *execution fragment* [2] as follows.

Definition 2.3. *An* execution fragment *of an interface automaton P is a possibly infinite, alternating sequence of states and actions $v_0, \alpha_0, v_1, \alpha_1, \ldots$ such that $(v_i, \alpha_i, v_{i+1}) \in T_P$.*

Definition 2.4. *Given an interface automaton $P = \langle V_P, V_P^{init}, A_P^{\mathcal{I}}, A_P^{\mathcal{O}}, A_P^{\mathcal{H}}, T_P\rangle$, an action $\alpha \in A_P^{\mathcal{I}}$ and a state $v \in V_P$, a* reaction sequence *to α in v is a possibly infinite trace of actions $\sigma = \alpha_0, \alpha_1, \ldots$ such that*

- *$\alpha_i \in A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}}$,*
- *$\exists v, v_0, v_1, \ldots \in V_P$ such that $v, \alpha, v_0, \alpha_0, v_1, \alpha_1, \ldots$ is an execution fragment of P and*
- *if σ has finite length n then $\forall \beta \in A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}} \ . \ v_n \overset{\beta}{\not\mapsto}$.*

We say that α is an activator *of σ in v and denote it with $v \underset{\alpha}{\overset{\sigma}{\Longrightarrow}} v_n$ if σ is finite or $v \underset{\alpha}{\overset{\sigma}{\Longrightarrow}} \uparrow$ otherwise.*

2.2. Trace validity

In this section we provide a formal definition of compliance of a trace with respect to a gate automaton. Intuitively, we can imagine that a sequence of actions is allowed by a gate automaton if, passing it as the input of the (instantiation of the) automaton, the output is the unchanged sequence. We formally define this notion in terms of reactions sequences in the following way.

Definition 2.5. *Given a finite trace of actions $\sigma = \alpha_1, \ldots, \alpha_n$ and a gate automaton $\mathcal{G} = \langle V, \imath, A, T \rangle$ we say that σ is* weakly compliant *with $\mathcal{G}$, in symbols $\sigma \vdash \mathcal{G}$, if and only if for any instantiation $\mathbf{G}_k$ of $\mathcal{G}$ we have $\imath \overset{\sigma_1^{k+1}}{\underset{\langle\alpha_1,k\rangle}{\Longrightarrow}} v_1 \ldots \overset{\sigma_n^{k+1}}{\underset{\langle\alpha_n,k\rangle}{\Longrightarrow}} v_n$ such that $\sigma_i^{k+1} = \langle\beta_{i,1}, k+1\rangle \ldots \langle\beta_{i,m_i}, k+1\rangle$ and $f_{out}(\sigma_1^{k+1} \ldots \sigma_n^{k+1}) = \sigma$ where f_{out} is the function recursively defined as*

$$f_{out}(\sigma\sigma') = f_{out}(\sigma)f_{out}(\sigma') \quad f_{out}(\langle\alpha, h\rangle) = \alpha \quad f_{out}(\langle\blacklozenge, h\rangle) = \cdot$$

being $\cdot$ the empty trace and $\blacklozenge \in \{\blacktriangle, \blacktriangledown\}$.

Beyond the technical definition, the weak compliance of a trace with respect to a gate automaton is quite intuitive. In particular, we can see the weak compliance as the dual of *transparency*. That is, a trace weakly complies with a gate automaton if and only if an external observer cannot understand whether the trace has been processed by (the instantiation of) the automaton or not.

Clearly, weak compliance does not correspond to a full transparency. Indeed, the transitions of the automaton can introduce and delete actions in such a way that a trace is kept unchanged as a whole, but its prefixes are modified. For characterising sequences that are not modified at all by a gate automaton we use the notion of *strong compliance*.

Definition 2.6. *Given a finite trace of actions $\sigma = \alpha_1, \ldots, \alpha_n$ and a gate automaton $\mathcal{G} = \langle V, \imath, A, T \rangle$ we say that σ is* strongly compliant *with $\mathcal{G}$, in symbols $\sigma \models \mathcal{G}$, if and only if for any prefix σ' of σ holds that $\sigma' \vdash \mathcal{G}$.*

2.3. Gate Automata and ConSpec

The *Contract Specification Language* [1], *ConSpec* for short, has been proposed as a formalism for defining both behavioural contracts and security policies. Roughly, the syntax of ConSpec resembles to the statements of an imperative programming language. Here we recall the syntax of ConSpec and we show how ConSpec specifications can be translated into corresponding gate automata. Note that, for simplicity, we omit few details of the original ConSpec syntax irrelevant for our purposes.

2.3.1. ConSpec syntax

Briefly, a ConSpec specification is composed by three blocks: (i) a preamble, (ii) a security state and (iii) a finite list of clauses. The preamble just declares the range of values for the used variables (`MAXINT` and `MAXLEN`)[1]. The security state is a list of variables declarations following the schema τ `x` $::=$ `v` where $\tau \in \{\texttt{bool}, \texttt{int}, \texttt{string}\}$ is a type, `x` is a variable name and `v` is a value of type τ. Note that here types are bounded, *i.e.*, they represent a finite number of values. For instance, if we set `MAXINT` to 3 then integer values range in $\{0, 1, 2, 3\}$.

Each clause contains a parametric action $\alpha(\tau\ y)$, activating the rule, and a list of conditional instructions. Action names belong to a denumerable set Λ, *i.e.*, $\alpha \in \Lambda$, and types are the same as for the security state. The left side of the conditional instructions is a decidable, boolean guard g defining a property of the security state and action parameter, while the right side is an update statement u (*i.e.*, a possibly empty block of variable assignments). We assume all the guards of a single clause to be pairwise disjoint, *i.e.*, if g and g' belong to the same clause then it never happens that $g \wedge g'$ is verified. Figure 1 shows the syntax described above. The

`MAXINT` n
`MAXLEN` m
 	
`SECURITY STATE`
τ_1 `x`$_1$ $::=$ `v`$_1$`;` $\cdots$ τ_N `x`$_N$ $::=$ `v`$_N$`;`
	`BEFORE` $\alpha_1(\tau'_1\ y_1)$ `PERFORM`
	g_1^1 `->` $u_1^1 \cdots g_{M_1}^1$ `->` $u_{M_1}^1$
	$\vdots$
	`BEFORE` $\alpha_K(\tau'_K\ y_K)$ `PERFORM`
	g_1^K `->` $u_1^K \cdots g_{M_K}^K$ `->` $u_{M_K}^K$

Figure 1: The syntax of ConSpec preamble, security state (left) and clauses (right).

structure of the security clauses needs a further dissertation. Indeed, comparing it with the standard one [1], we see two main differences: (i) we only have before-event checks (*i.e.*, we do not use the keywords `AFTER` and `EXCEPTIONAL`) and (ii) we use monadic actions. We claim that these simplifications do not reduce the expressive power of the ConSpec language. As a finite number of parameters can be encoded in a single one, using monadic actions is not a restriction. For instance, we could use strings to encode n-arguments actions (*e.g.*, $\alpha(\text{"}3, \texttt{msg}, \texttt{false}\text{"})$ for $\alpha(3, \text{"}\texttt{msg}\text{"}, false)$). For the sake of simplicity, we refer to this encoding in our examples and we assume to have the functions $getPar\_\tau$: `int` $\times$ `string` $\rightarrow \tau$,

[1]The standard ConSpec syntax also contains statements defining the scope of a policy, *i.e.*, `Session`, `Multisession` and `Global`. However, it is immaterial for our purposes and we can simply neglect it.

such that $getPar\_\tau(i, s)$ returns the i-th parameter (having type τ) encoded in s. Also, we require all the variable and parameter names to be unique and all the clauses to be triggered by different actions.

Moreover, we can simulate the behaviour of `AFTER` and `EXCEPTIONAL` clauses by introducing new actions. As a matter of fact, the standard syntax of ConSpec is oriented to model the computations of object-oriented systems, *i.e.*, passing through method invocations. Every method triggers the clauses when it is invoked, when it returns a result and, possibly, when raising an exception. Then, for each method `m` we can define three actions $\alpha^B_{\mathtt{m}}$, $\alpha^A_{\mathtt{m}}$ and $\alpha^E_{\mathtt{m}}$ representing the method invocation, standard return and exceptional return, respectively.

Example 2.1. *Consider the policy saying "An application cannot open connections after reading local files". We model the involved methods through the actions* `fopen(int mode)` *and* `copen(string url)`. *Where* `mode` $\in \{0, 1, 2, 3\}$ *is a two-bits mask representing the access type (*i.e.*, 00 = none, 01 = read, 10 = write and 11 = read and write), and* `url` *is a network address. The resulting policy is:*

```
MAXINT 3
MAXLEN 0
SECURITY STATE
bool accessed ::= false;

BEFORE fopen(int mode) PERFORM
  (mode == 1) -> {accessed ::= true;}
  (mode == 3) -> {accessed ::= true;}
  (mode == 0 || mode == 2) -> {}
BEFORE copen(string url) PERFORM
  !accessed -> {}
```

2.3.2. Gate automata interpretation of ConSpec

The semantics of ConSpec can be interpreted using gate automata. Given a state q and a guard g, we say that g is valid in q ($q \vdash g$) if and only if replacing the variable names of g using the mapping defined by q we obtain a tautology. Moreover, we say that an update block u denotes a function, namely $[\![u]\!]$, from states to states, *i.e.*, $[\![u]\!] : Q \to Q$.

We obtain a gate automaton from a ConSpec specification as follows.

States. The set Q of states is fully characterised by the security states and the actions parameters. In particular, we define a state q as a mapping from variable and parameter names to the lifted domain of possible values. Formally, given a variable or parameter name x, then $q(x) = v$ with $v \in Val \cup \{\bot\}$ (where $Val = \mathtt{int} \cup \mathtt{bool} \cup \mathtt{string}$). Moreover, to be valid a state must assign to each variable a value different from $\bot$ and to at most one parameter a value that is different from $\bot$. Hence, Q is the set of all the possible, valid combinations of

assignments. Note that, as ConSpec uses bounded types, the number of states is always finite.

Initial state. The initial state $\imath \in Q$ is the set mapping the variables of the security state to their initial values and the parameters to the undefined, $\perp$ value.

Alphabet. The set of events A that the automaton can read is the set of pairs $\{\langle \alpha, v\rangle \mid \alpha \in \Lambda \wedge v \in Val\}$. We use $\alpha(v)$ instead of $\langle \alpha, v\rangle$ where unambiguous.

Transitions. We build the set T of transitions in the following way. For each ConSpec clause we take the triggering action $\alpha(\tau\ x)$ and we list all the states $q \in Q$ such that $q(x) = \perp$. Then we proceed as follows.

1. For each possible event $\alpha(v)$ we add a transition from $q \xrightarrow{\bar{\alpha}(v)} q'$, where $\forall y \neq x.q'(y) = q(x)$ and $q'(x) = v$.
2. For each conditional instruction $g \rightarrow u$ of the clause and for each of the freshly added transitions, if $q' \vdash g$ then we add a transition $q' \xrightarrow{\alpha(v)} [\![u]\!](q)$.
3. For all the states $\dot{q}$ such that $x = \perp$ and for all the events $\alpha(\dot{v})$ such that $\dot{q} \not\xrightarrow{\bar{\alpha}(\dot{v})}$, we add a transition $\dot{q} \xrightarrow{\bar{\alpha}(\dot{v})} \dot{q}$.

We iterate these steps until every clause has been processed.

Example 2.2. *We create a gate automaton for the specification in Example 2.1.*

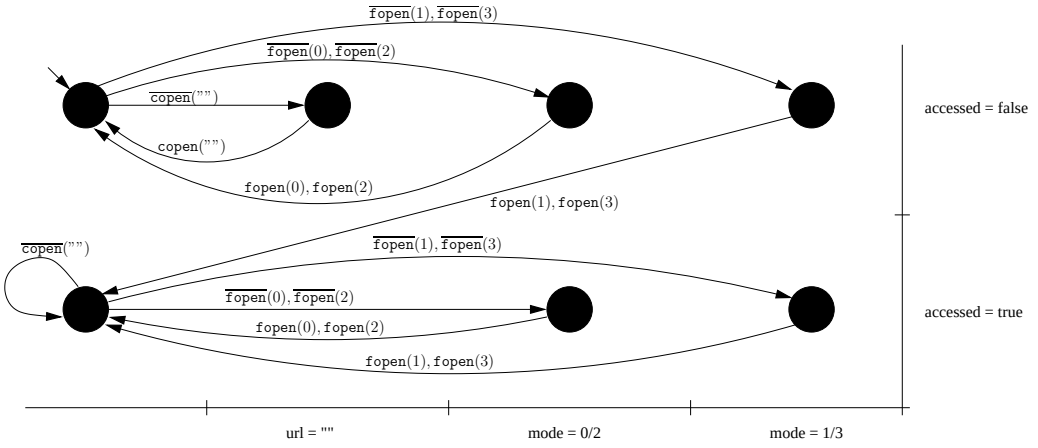

Figure 2: The conversion of a ConSpec specification into a gate automaton.

Figure 2 shows the gate automaton produced by the procedure described above. Rows and columns denote the values of variables for the automaton states, for instance the top row contains the states q such that $q(\texttt{accessed}) = \texttt{false}$. The leftmost column contains the states assigning no values to the actions parameters.

The unreachable state in position $\texttt{accessed} = \texttt{true}, \texttt{url} = ""$*, which would correspond to a specification violation, has been removed. Also, two pairs of column,* i.e., $\texttt{mode} = 0/2$ *and* $\texttt{mode} = 1/3$*, have been grouped as their states share the same behaviour. Finally, we did not draw immaterial self loops,* i.e., *representing transitions that cannot take place.*

Clearly, the procedure described above can be optimised in several ways, *e.g.*, removing unreachable states or collapsing groups of equivalent states. Nevertheless, our purpose is to show that gate automata can be suitably used to encode ConSpec policies and contracts.

The following property ensures that gate automata correctly encode ConSpec specifications.

Property. Given a ConSpec specification S and the gate automaton $\mathcal{G}$ obtained through the procedure defined above, then a trace σ complies with S if and only if it also complies with $\mathcal{G}$.

Proof. (Sketch) Intuitively, we build the ConSpec automaton for S as described in [1]. Then, we show that there exists a bijective mapping among the states and the transitions of the ConSpec automaton and the those of the gate automaton. Finally, we proceed by induction on the length of σ, showing that the outputs of the two automata reading σ is the same. □

The previous property guarantees that gate automata can be suitably used for implementing ConSpec-based security frameworks. In the next section we exploit this property for defining a security enforcement model.

3. Implementing the S×C×T runtime through Gate Automata

The S×C×T has been originally presented in [6, 5] as a unique framework for managing both security and trust in a computing environment. It uses two behavioural specifications: the *contract* of an application and the *policy* of the hosting platform. Intuitively, a contract declares and exhaustively describes the possible behaviours of an application. Instead, a policy represents all the behaviours that the execution environment will accept as legal from a running program. Usually, the application vendors provide the contracts while the platform owners/administrators define the policies.

The S×C×T workflow, depicted in Figure 3, shows the two phases of the application deployment process: the trustworthiness evaluation and the assignment to a security domain. When an application enters the deployment procedure, *i.e.*, before its first execution, the trust module decides about the trustworthiness of the code provider. This amounts to accept the trustworthiness of the contract and its source.

If this check is not passed, *i.e.*, the system rejects the vendor's trustworthiness, then the application runs in the scope of the *policy enforcement* mechanism. Otherwise, if the trust check successes, the system checks whether the contract complies with the security policy. In case of compliance, the system executes the application under a *contract monitoring* setting. While the policy enforcement process prevents the security violations, the monitoring facility keeps under control the possible contract violations. When a running program violates its contract, *i.e.*, it tries to behave in an undeclared way, the system reacts by changing the trust level of the application provider.

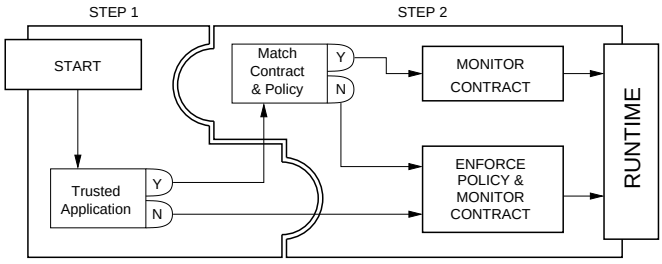

Figure 3: The Security-by-Contract-with-Trust Workflow.

Here we introduce an implementation of the S×C×T runtime support using gate automata. According to the S×C×T standard model [5], applications run in the scope of one of the two security domains described above. In both cases, running programs are dynamically checked for compliance with respect to their contract (*i.e.*, contract monitoring process). Moreover, the applications watched by the policies enforcement facility are checked for possible policy violations.

The platform owners declare their security policies through gate automata either directly or translating ConSpec policies (see Section 2.3.2). Instead, we assume that the contracts are always specified through ConSpec.

Starting from a ConSpec contract, we build a corresponding gate automaton by following the procedure for policies presented in the previous section. The only difference is that here we replace the third step of the transitions creation procedure with

3. For all states $\dot{q}$ s.t. $x = \perp$ and for all events $\alpha(\dot{v})$ s.t. $\dot{q} \stackrel{\bar{\alpha}(\dot{v})}{\not\longrightarrow}$, we add a fresh, new state $q^\star$ in Q and a pair of transitions $\dot{q} \stackrel{\bar{\alpha}(\dot{v})}{\longrightarrow} q^\star$ and $q^\star \stackrel{\blacktriangledown}{\rightarrow} \dot{q}$ in T.

In this way, as expected, a contract violation leads to a trust penalty. This behaviour implements the S×C×T reaction to the contract violations.

We use the gate automata specifications of policies and contracts for implementing the S×C×T runtime environment. We consider a program R as a source

of the security-relevant actions, which are the side effects of the programs' executions. Moreover, we assume the enforcement environment to be effective, *i.e.*, R can be suspended before the actual execution of the operation corresponding to the ongoing action. For instance, if R tries to access a resource, so raising an access action, it actually obtains the permission only after checking the security settings.

The first component of the enforcement environment is the *trust management system* (TMS). This component handles the trust weights associated to each agent and provides an implementation of the two internal actions ▲ and ▼. While following the execution of its target, the enforcement environment can perform one or more actions of type ▲ and ▼. The TMS receives these signals and increases (decreases) the target trust level. Note that some TMSs use a finer characterisation of rewards and penalties, *i.e.*, more than two actions. Nevertheless, this behaviour is fully compatible with our model. Indeed, we can easily extend the set of internal actions or simulate it by adding more consecutive transitions.

The enforcement environment also contains a set of gate automata $\mathcal{G}^1, \ldots, \mathcal{G}^n$ composing the *policy pool* (PP). The automata in the policy pool are associated to a certain level of trust $0 \leqslant t \leqslant 1$ on which they are inversely ordered, *i.e.*, $1 \leqslant i < j \leqslant n$ implies that $t_i > t_j$. We also insert the gate automaton obtained from the contract of R in PP. The level of trust of this automaton is always equal to 1 and it is the first in the ordering.

When a target R, having trust level t_R, starts its execution, the policy pool instantiates all the gate automata $\mathcal{G}^i$ such that $t_i \geqslant t_R$ to the corresponding interface automata $\mathbf{G}_i^i$ (see Section 2). Then, the resulting interface automata are composed to create an *interface automata stack* which is applied to R. Note that the automaton obtained from the contract of R is always in the first position of the stack, *i.e.*, the stack bottom.

The stack receives the actions performed by R and processes them by passing the reaction sequences of each automaton to the layer above. More in detail, assuming that the current state of each interface automaton $\mathbf{G}_i^i$ is v_i, every layer of the stack follows this procedure:

1. $\mathbf{G}_i^i$ receives a trace σ^i from the level below;
2. for each element $\langle \bullet, i \rangle$ of σ^i execute the following sub steps:
   (a) if $\bullet =$ ▲ (▼) then require the TMS to increase (decrease) t_R.
   (b) otherwise, if $\bullet = \alpha$ compute $v_i \overset{\sigma^{i+1}}{\underset{\langle \alpha, i \rangle}{\Longrightarrow}} v_i'$ and pass the control to the layer above (by invoking this procedure);
3. return the control to the level below.

When R fires some action α, the previous steps are executed starting from the first layer, representing the contract of R, with $\sigma^1 = \langle \alpha, 1 \rangle$. The output of the last layer

(after removing the index k) is a sequence of reactions that have been stimulated by α, that is, the enforcement result.

As the actions pass through the stack levels, the TMS receives trust adjustment signals. As a consequence, the TMS updates t_R, possibly causing the system to add or remove one or more automata in the stack.

4. Related Work

Some works about the integration between trust management and security enforcement are present in the literature. However, few of them deal with the mobile applications. Koshutanski et al. [12] present an access control system enhancing the Globus toolkit standard support. Their proposal copes with the performances issues arising from the access rights management support of Globus for shared resource in the Grid architecture. Along this line of research [4] presents an integrated architecture, extending the previous one, with an inference engine managing reputation and trust credentials. This framework is extended again in [11] where a trust credentials negotiation module is introduced to overcome some scalability problems. In this way, the new framework guarantees the privacy credentials and the security policies of both users and providers. Even though the application scenario and the implementation are different, the basic idea consists of a trust-based metrics used for deciding about the reliability of an application provider.

The automata-based specification of security policies has a long-standing tradition. In [16], the author advocates security automata for defining security requirements and for implementing the corresponding controllers. We can observe that gate automata extend the automata of [16] in two ways: (i) they can add and remove actions from the target's execution trace (rather than simply halt it) and (ii) they also use special actions for the trust management. For this reason, in [8] we showed that gate automata can be encoded using *edit automata* [13]. Gate automata differ from edit automata mainly because they manage trust. Indeed, they integrate the trust management process and the enforcement mechanism in a unique model. Moreover, they inherit the compositionality properties of interface automata [2]. Hence, reasoning about the composition of gate automata is generally simpler than for edit automata.

Also [3] proposes an automaton-based specification, *i.e.*, *usage automata*, of security policies, *i.e.*, *usage policies*. Usage automata slightly differ from security automata. Roughly, an execution trace complies with a usage policy iff it is not accepted as an input word by the corresponding usage automaton. Moreover, usage policies are applied directly to the source code through proper syntactical operators that also causes the composition of policies through scope nesting. Again, the main differences with respect to our automata are that (i) usage automata do not change the observed trace and (ii) they do not handle trust. Furthermore, in the

environment using gate automata the scope of a policy is not fixed but policies are activated/deactivated according to the trust values.

In [14] the authors present a method for modelling security automata through process algebra operators. They extends some existing results on process algebras to the analysis, verification and synthesis of secure systems. Also in [10, 15] a process algebra-based language, namely *POLPA*, is used for policy specification and enforcement. In general, process algebras are more expressive than finite state automata. However, these works propose no integration between security and trust.

5. Conclusion and Future Work

In this paper we presented *gate automata* for specifying integrated security and trust policies. We also compared our proposal with ConSpec showing that gate automata can be suitably used for specifying both policies and contracts. Finally, we proposed an implementation model for the S×C×T runtime support.

As future work, we aim at investigating model checking techniques for gate automata. This will extend the present work with the static verification module necessary for a full implementation of the S×C×T. We are also interested in the theoretical aspects of the parallel composition of gate automata. Indeed, the current enforcement environment uses a stack-based composition. This structure do not take into account concurrency. Hence, we would like to study the possibility of composing two or more automata stacks for extending our model to concurrent programming models.

The implementation of a prototype is currently under investigation. In [5] we presented simulation results showing the feasibility of our trust management strategy. In particular, we showed that our proposal implementation rapidly converges when some attacks take place. In [7] and [9] two enforcement environment using ConSpec have been introduced. Both these implementations have good performances and guarantee the feasibility of the enforcement method to which we are aligned. In our opinion, these results represent more than optimistic premises for our model.

References

[1] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. *Sci. Comput. Program.*, 74(1-2):2–12, 2008.

[2] L. de Alfaro and T.A. Henzinger. Interface automata. In ACM, editor, *Proceedings of the 8th European software engineering conference*, 2001.

[3] M. Bartoletti. Usage automata. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, volume 5511 of *LNCS*, pages 52–69, 2009.

[4] M. Colombo, F. Martinelli, P. Mori, M. Petrocchi, and A. Vaccarelli. Fine grained access control with trust and reputation management for globus. In *OTM Conferences (2)*, pages 1505–1515, 2007.

[5] G. Costa, N. Dragoni, V. Issarny, A. Lazouski, F. Martinelli, F. Massacci, I. Matteucci, and R. Saadi. Security-by-Contract-with-Trust for mobile devices. *JOWUA*, 1(4):75–91, Dec. 2010.

[6] G. Costa, N. Dragoni, A. Lazouski, F. Martinelli, F. Massacci, and I. Matteucci. Extending Security-by-Contract with quantitative trust on mobile devices. In *Proceedings of CISIS 2010*, pages 872–877, 2010.

[7] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter. Runtime monitoring for next generation Java ME platform. *Computers & Security*, July 2009.

[8] G. Costa and I. Matteucci. Trust-driven policy enforcement through gate automata. In *Proceeding of IMIS 2011*. Accepted for pubblication.

[9] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The s3ms.net run time monitor: Tool demonstration. *Electronic Notes in Theoretical Computer Science*, 253(5):153–159, 2009.

[10] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, and E. Vetillard. Security-by-Contract (SxC) for software and services of mobile systems. In *At your service - Service-Oriented Computing from an EU Perspective*. MIT Press, 2008.

[11] H. Koshutanski, A. Lazouski, F. Martinelli, and P. Mori. Enhancing grid security by fine-grained behavioral control and negotiation-based authorization. *Int. J. Inf. Sec.*, 8(4):291–314, 2009.

[12] H. Koshutanski, F. Martinelli, P. Mori, L. Borz, and A. Vaccarelli. A fine grained and x.509 based access control system for globus. In *OTM Conferences (2)*, pages 1336–1350. Springer, 2006.

[13] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Int. J. Inf. Secur.*, 4(1–2), Feb. 2005.

[14] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *ENTCS*, 179:31–46, 2007.

[15] F. Martinelli and P. Mori. On usage control for grid systems. *In Future Generation Computer Systems. Elsevier Science*, 26(7):1032–1042, 2010.

[16] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.