Service Automata

Richard Gay, Heiko Mantel, and Barbara Sprick

Modeling and Analysis of Information Systems,
Department of Computer Science, TU Darmstadt, Germany
{gay,mantel,sprick}@mais.informatik.tu-darmstadt.de

Abstract. We propose a novel framework for reliably enforcing security in distributed systems. Service automata monitor the execution of a distributed program and enforce countermeasures before a violation of a security policy can occur. A key novelty of our proposal is that security is enforced in a decentralized though coordinated fashion. This provides the basis for reliably enforcing global security requirements without introducing unnecessary latencies or communication overhead. The novel contributions of this article include the concept of service automata and a generic formalization of service automata in CSP. We also illustrate how the generic model can be tailored to given security requirements by instantiating its parameters in a stepwise and modular manner.

1 Introduction

If the security of a program cannot be certified a priori, then one can establish trustworthiness a posteriori by encapsulating the program with a runtime monitor. The monitor checks if the program's actions comply with a given security policy and modifies the program's behavior when a policy violation is about to occur. There are various approaches to implement such security monitors, e.g., by in-lining them into the program code or by integrating them into the run-time environment. For verifying the soundness of an implementation, the intended behavior of the monitoring framework can be captured in an abstract, formal model. Naturally, such a security model can also be used for proving implementation-independent properties of a monitoring framework.

In this article, we propose *service automata* as a novel framework for enforcing security requirements at runtime and present a formal security model for this framework. Service automata are parametric in the security policy and can be used for enforcing a wide range of security requirements. That is, our approach is in the tradition of generic security monitoring that began with security automata [14] and that has gained much popularity since (see, e.g., [7,10,4,11,3]).

Our objective is to lift generic security monitoring to distributed systems. The distinctive feature of service automata over prior frameworks is that they support decentralization of monitoring and enforcement in a coordinated fashion. In comparison to a centralized approach, where security is enforced by a dedicated node of the distributed system, service automata allow one to reduce communication overhead and latencies. In comparison to a fully decentralized approach, where

Published in:
G. Barthe, A. Datta, and S. Etalle (Eds.): FAST 2011, LNCS 7140, pp. 148–163, 2012.

each program is encapsulated by a monitor that enforces a local policy, service automata are more expressive because they can also enforce non-local security requirements (such as, e.g., separation of duty or Chinese Walls) by using their communication capabilities. However, if desired, fully centralized as well as fully decentralized enforcement can also be realized in our framework.

A technical novelty is that service automata themselves have a modular architecture. This creates the possibility to instantiate service automata in a stepwise manner, which we find particularly attractive. Firstly, it reduces conceptual complexity because aspects such as enforcement, delegation, and coordination can be addressed separately when defining an instantiation. Secondly, it enables the re-use of components of an instantiation (e.g., when modifying coordination while leaving enforcement and delegation unchanged). One could even envision a library of commonly used parameters for service automata.

In summary, the main novel contributions of this article are:

- the concept of service automata for decentralized and coordinated enforcement of security in distributed systems (Section 2);
- a generic, formal security model for service automata in Hoare's CSP [9] being parametric in the program, the security policy, and the enforcement component (Section 3); and
- examples for how the formal model of service automata can be instantiated to soundly enforce given security requirements (Sections 4 and 5).

2 Service Automata – the Concept

Runtime monitors for security provide protective encapsulations of programs, possibly working in more than one direction. They can protect the environment against misbehavior as well as malfunctioning of a program, and they can protect the program against illegitimate or unforeseen input by the environment.

Our novel concept of service automata enables generic security monitoring in distributed systems such as service-oriented architectures (hence the name service automata). The primary goal of our proposal is not to increase expressiveness (i.e. the class of security properties that can be enforced), but rather to enable decentralized enforcement in a coordinated fashion. We aim for the avoidance of bottlenecks and a reduction of communication overhead, i.e. the typical drawbacks of centralized security monitoring. In comparison to prior approaches for decentralized monitoring, we aim for the sound enforcement of a wider range of security aspects, including ones that cannot be decided locally at a node in a distributed system (see Section 6 for a detailed comparison with related work).

An individual *service automaton* supervises the execution of a single program at some node of a distributed system in order to enforce a given policy. Whenever the local program is about to execute an action that might be relevant for the policy then this action is intercepted, and the execution of the program is temporarily blocked. The service automaton then determines whether the action complies with the policy and either permits the action or takes countermeasures to enforce the policy. Possible countermeasures include terminating the program,

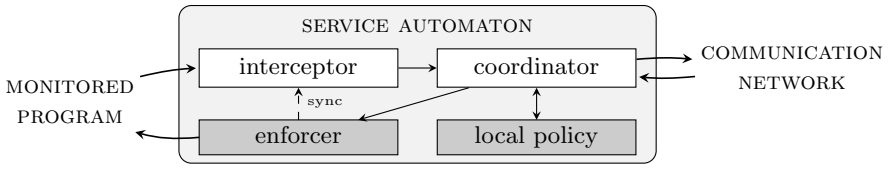

Fig. 1. Interfaces, internal structure, and parameters of a service automaton

skipping the problematic action, or executing additional or alternative actions. That is, the countermeasures against policy violations correspond to the ones of edit automata [10] (and, hence, go beyond the ones of security automata [14]).

The key novelty of our concept is that a service automaton can communicate with other service automata in a distributed system. This communication capability is crucial for decentralizing security monitoring and enforcement to a large extent while still being able to enforce non-local security aspects.

Each service automaton has a modular architecture (see Figure 1), consisting of four components: the *interceptor* that intercepts the respectively next security-relevant action of the program, the *coordinator* that determines whether the action complies with the *local (security) policy* and decides upon possible countermeasures, and the *enforcer* that implements these decisions.

Two of these components (the local policy and the enforcer) are left parametric in the definition of service automata (indicated by the gray boxes in the figure). They have to be instantiated when applying service automata.

The coordinator uses the local policy to make decisions and the enforcer to impose those decisions onto the monitored program. If the local policy is not sufficient to decide whether a given event may occur, then the coordinator may delegate the decision to some other service automaton. Conversely, the coordinator might receive delegation requests from other service automata and resolve them on their behalf. In order to support delegation in distributed systems where the nodes are not fully connected, coordinators on intermediate nodes need to also support routing of delegation requests and of corresponding responses.

Obviously, the concept of service automata can be implemented in various ways and formal security models can be specified using many specification formalisms. For the remainder of this article, we choose Hoare's CSP as specification formalism for our formal security model and abstract from implementation issues.

3 Service Automata – a Formal Model

3.1 A Primer to Hoare's Communicating Sequential Processes

We briefly recall the sublanguage of Hoare's Communicating Sequential Processes (CSP) used in this article. For a proper introduction, we refer to [9].

A *process* P is a pair $(\alpha(P), \mathit{traces}(P))$ consisting of a set of events and a nonempty, prefix-closed set of finite sequences over $\alpha(P)$. The *alphabet* $\alpha(P)$ contains all events in which P could in principle engage. The *set of possible traces*

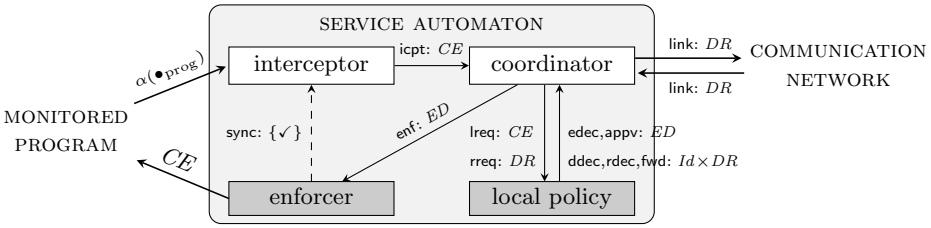

Fig. 2. Communication interface between service automata components

$traces(P) \subseteq (\alpha(P))^*$ contains all sequences of events that the process could in principle perform. We use $\langle\,\rangle$ to denote the empty sequence, $\langle e \rangle$ to denote the trace consisting of the single event e, and $s.t$ to denote the concatenation of two traces s and t. That an event e occurs in a trace t is denoted by $e \lhd t$.

The CSP process expression STOP_E specifies a process with alphabet E and a set of traces containing only $\langle\,\rangle$. A process that performs event e and then behaves according to the process expression P, is specified by $e \to P$. External and internal choice between P and Q are specified by $P \,\Box\, Q$ and $P \sqcap Q$, respectively. They model that the process behaves according to either P or Q. The parallel composition of P and Q is specified by $P \parallel Q$. The parallel processes have to synchronize on the occurrences of all events that their alphabets have in common. The process $P \setminus E$ behaves as P but all events in the set E are hidden by removing them from the process' alphabet and possible traces.

The binary operators $\Box$, $\sqcap$ and $\parallel$ are lifted to n-ary operators over non-empty finite index sets. For instance, $\Box_{x \in X} P(x)$ equals $P(a)$ if $X = \{a\}$ and equals $P(a) \,\Box (\Box_{x \in X \setminus \{a\}} P(x))$ if $a \in X$ and X contains at least two elements.

We use structured events of the form $\mathsf{c}.m$ to model the communication of a message m on a channel c. In a process expression we write $\mathsf{c}!m$ instead of $\mathsf{c}.m$ in order to indicate that message m is sent on c, and use $\mathsf{c}?x\colon M$ for receiving some message $m \in M$ on channel c while instantiating the variable x with m. Effectively, $\mathsf{c}?x\colon M \to P(x)$ corresponds to an external choice on the events in $\{c.m \mid m \in M\}$ such that the computation continues according to $P(m)$.

A process definition $\mathsf{NAME} \stackrel{\text{def}}{=}_{\alpha} P$ declares a new process name NAME and defines that NAME models a process whose traces are given by the process expression P and whose alphabet equals α. We omit the subscript α in a process definition if the alphabet of NAME shall equal $\alpha(P)$. Process names can be used as subexpressions within process expressions, thus allowing for recursion.

Properties of CSP processes are modeled by unary predicates on traces. We say that a unary predicate φ on traces is *satisfied* by a process P (denoted by $P \,\mathsf{sat}\, \varphi$) if and only if $\varphi(t)$ holds for each $t \in traces(P)$.

3.2 The Generic Model of Service Automata

The formal model of a single security automaton reflects the modular architecture introduced in Section 2. We will model a service automaton as the parallel

composition of an interceptor process INT, a coordinator process COR, a local policy $\bullet_{\mathrm{pol}}$ and an enforcer $\bullet_{\mathrm{enf}}$. These components of a service automaton interact with each other via unidirectional communication channels, as depicted in Figure 2. The interceptor sends messages to the coordinator via channel icpt. The coordinator sends messages to the local policy on channels lreq and rreq and receives responses of different kinds on the channels edec (enforcement decisions), ddec (delegation decisions), rdec (remote decisions), fwd (forwarded messages), and appv (approvals of remote decisions). The coordinator also sends messages to the enforcer on channel enf and the enforcer unblocks the interceptor via channel sync. We also use unidirectional channels for the communication between different service automata. The channel $\mathsf{link}_{i,j}$ is used by the coordinator of the service automaton with identifier i to send messages to the coordinator of service automaton j. We assume that the set Id of all *identifiers* of service automata of a given distributed system is finite.

In this section, we present the process definitions for the interceptor INT, the coordinator COR, and the service automaton SA. The enforcer and the local policy are parameters in the definition of SA that have to be instantiated when applying service automata (see Definition 1 and Sections 4 and 5).

Interceptor. The interceptor specification is parametric in α, the alphabet of the monitored program, and in β, the set of security-critical events of this program:

$$\begin{aligned}\mathsf{INT}_i(\alpha,\beta) \stackrel{\text{def}}{=} & \Box_{ev\in\alpha\setminus\beta}\, ev \to \mathsf{INT}_i(\alpha,\beta)\\ & \Box\, \Box_{ev\in\alpha\cap\beta}\, ev \to \mathsf{icpt}!ev \to \mathsf{sync}?x\colon \{\checkmark\} \to \mathsf{INT}_i(\alpha,\beta)\end{aligned}$$

The interceptor synchronizes with the monitored program on each event $ev \in \alpha$. If ev is not policy-relevant (first line) then the interceptor simply awaits the next event. If ev is policy-relevant (second line) then the interceptor sends ev to the coordinator via channel icpt. It then waits until it is unblocked by the enforcer via sync. This synchronization ensures that the interceptor and the monitored program can only proceed after a decision about ev has been made and enforced.

Enforcer. The enforcement of decisions is not specified in this section as the enforcer is a parameter of the generic model. Here, we only assume a set γ of *enforcement decisions* that the enforcer is willing to accept from the coordinator. Moreover, we expect instantiations of the enforcer to properly unblock the interceptor (and thereby also the monitored program) via sync such that they can proceed. We present definitions of typical enforcers in Section 4.1.

Coordinator. The coordinator specification is parametric in β, the set of security-critical events of the locally monitored program, and in γ, the set of enforcement decisions. Moreover, it assumes a set DR of *delegation decisions* and *delegation responses* that is identical for all service automata in a given system:

$$
\begin{aligned}
\mathsf{COR}_i(\beta,\gamma) \stackrel{\text{def}}{=}\ & \mathsf{icpt}?\,ev\colon \beta \to \mathsf{lreq}!\,ev \\
& \to \Big(\big(\mathsf{edec}?\,ed\colon \gamma \to \mathsf{enf}!\,ed \to \mathsf{COR}_i(\beta,\gamma)\big) \\
& \quad \Box\ \big(\mathsf{ddec}?(k,dr)\colon (Id\backslash\{i\})\times DR \to \mathsf{link}_{i,k}!\,dr \to \mathsf{COR}_i(\beta,\gamma)\big)\Big) \\
& \Box \quad \Box_{j\in Id\backslash\{i\}}\ \mathsf{link}_{j,i}?\,dr\colon DR \to \mathsf{rreq}!\,dr \\
& \to \Big(\big(\mathsf{fwd}?(k,dr')\colon (Id\backslash\{i\})\times DR \to \mathsf{link}_{i,k}!\,dr' \to \mathsf{COR}_i(\beta,\gamma)\big) \\
& \quad \Box\ \big(\mathsf{rdec}?(k,dr')\colon (Id\backslash\{i\})\times DR \to \mathsf{link}_{i,k}!\,dr' \to \mathsf{COR}_i(\beta,\gamma)\big) \\
& \quad \Box\ \big(\mathsf{appv}?\,ed\colon \gamma \to \mathsf{enf}!\,ed \to \mathsf{COR}_i(\beta,\gamma)\big)\Big)
\end{aligned}
$$

The coordinator receives an intercepted event ev from the interceptor (via icpt) or a delegation request/response dr from another service automaton (via $\mathsf{link}_{j,i}$).

In the first case, the coordinator passes ev to the local policy (via lreq). In response, the coordinator either receives an enforcement decision ed (via edec) that it passes on to the enforcer (via enf), or the coordinator receives a destination k and a delegation request dr (via ddec) and passes dr on to k (via $\mathsf{link}_{i,k}$).

In the second case, the coordinator receives a delegation request or delegation response dr from some other service automaton j (via $\mathsf{link}_{j,i}$) and passes it on to the local policy (via rreq). The coordinator then may receive a destination k and a (possibly modified) delegation request or response dr' (via fwd) that it then forwards to k (via $\mathsf{link}_{i,k}$). In addition, if dr is a delegation request, the coordinator receives a delegation response dr' (incorporating an enforcement decision) together with a destination k from the local policy (via channel rdec) and passes it to k via channel $\mathsf{link}_{i,k}$. If dr is a delegation response then the coordinator receives a local enforcement decision ed from the local policy (via appv) that it then forwards to the enforcer (via enf).

Service Automaton. Our CSP specification of a service automaton with identifier i is modular and parametric in the locally monitored program ($\bullet_{\text{prog}}$), the set of critical events of this program (β), the specification of the local policy ($\bullet_{\text{pol}}$), the set of enforcement decisions (γ), and the specification of the enforcer ($\bullet_{\text{enf}}$):

$$
\mathsf{SA}_i(\bullet_{\text{prog}},\beta,\bullet_{\text{pol}},\gamma,\bullet_{\text{enf}}) \stackrel{\text{def}}{=} \left[\begin{array}{ll} (\bullet_{\text{prog}} \parallel \mathsf{INT}_i(\alpha(\bullet_{\text{prog}}),\beta)) \setminus \beta & \\ \parallel & \bullet_{\text{pol}} \parallel (\mathsf{COR}_i(\beta,\gamma) \parallel \bullet_{\text{enf}}) \end{array}\right] \setminus H, \quad (1)
$$

Note that the structure of the above formal specification of a service automaton reflects the architecture depicted in Figure 1. The monitored program ($\bullet_{\text{prog}}$), the interceptor (INT_i), the coordinator (COR_i), the local policy ($\bullet_{\text{pol}}$), and the enforcer ($\bullet_{\text{enf}}$) are composed in parallel. In the definition of SA_i, most events of a service automaton are hidden using the set H that is defined as follows:

$$
H := \left\{ \begin{array}{l} \mathsf{sync}.\checkmark, \mathsf{icpt}.ev, \mathsf{enf}.ed, \mathsf{lreq}.ev, \mathsf{rreq}.dr, \\ \mathsf{edec}.ed, \mathsf{appv}.ed, \mathsf{ddec}.(k,dr), \mathsf{rdec}.(k,dr), \\ \mathsf{fwd}.(k,dr) \end{array} \middle| \begin{array}{l} ev \in \beta, ed \in \gamma, \\ dr \in DR, \\ k \in Id \setminus \{i\} \end{array} \right\} \quad (2)
$$

The set H contains all events used by the components INT_i, COR_i, $\bullet_{\text{pol}}$, and $\bullet_{\text{enf}}$ to communicate with each other. Hiding this set of events in (1) ensures that

the environment cannot interfere with the internal communication of a service automaton. When instantiating $\bullet_{\mathrm{pol}}$ and $\bullet_{\mathrm{enf}}$, we will ensure that the environment also cannot interfere with the logic of these components (see Definition 1).

The hiding of β in (1) enables the interceptor to learn about the next security-critical event that the locally monitored program is about to execute without making such events visible to the outside. Only the enforcer can cause critical events such that they are visible to the environment of the service automaton (note that $\bullet_{\mathrm{enf}}$ occurs outside the scope of the hiding operator for β). This use of hiding only becomes possible because service automata have a modular structure. Hiding β selectively allows us to monitor and to control security-critical actions of a monitored program before they can have an effect on the environment without having to transform the program (e.g. by renaming all security-critical events), which appears unavoidable with monolithic monitor specifications.

In order to enforce security requirements in a distributed system with service automata, one needs to instantiate the generic model with the components of the system that shall be encapsulated and six further parameters.

Definition 1. *A* service automata framework *is the process expression*

$$\|_{i \in Id} \mathsf{SA}_i(\mathsf{PRG}_i, CE_i, \mathsf{POL}_i, ED_i, \mathsf{ENF}_i)$$

that is uniquely determined by an instantiation

$$(Id, (\mathsf{PRG}_i)_{i \in Id}, (CE_i)_{i \in Id}, (\mathsf{POL}_i)_{i \in Id}, (ED_i)_{i \in Id}, DR, (\mathsf{ENF}_i)_{i \in Id}),$$

where Id is the finite set of all identifiers of monitored components and for each component $i \in Id$, PRG_i is the process expression specifying the component's behavior, CE_i is the component's set of critical events, POL_i is the process expression specifying the component's local security policy, ED_i is the component's set of possible enforcement decisions, DR is the set of possible delegation requests and responses, and ENF_i is the process expression specifying the enforcer.

We say that an instantiation is proper, *iff for all $i \in Id$ the following holds:*

(a) $CE_i \subseteq \alpha(\mathsf{PRG}_i)$,
(b) $\alpha(\mathsf{ENF}_i) = \{\mathsf{sync}.\checkmark\} \cup \{\mathsf{enf}.ed \mid ed \in ED_i\} \cup CE_i$,
(c) $\alpha(\mathsf{POL}_i) = \left\{ \begin{array}{l} \mathsf{lreq}.ev, \mathsf{rreq}.dr, \mathsf{edec}.ed, \mathsf{appv}.ed, \\ \mathsf{ddec}.(k, dr), \mathsf{rdec}.(k, dr), \mathsf{fwd}.(k, dr) \end{array} \middle| \begin{array}{l} ev \in CE_i, ed \in ED_i, \\ dr \in DR, k \in Id \setminus \{i\} \end{array} \right\}$,
(d) $\alpha(\mathsf{PRG}_i) \cap \left(\begin{array}{l} \alpha(\mathsf{COR}_i(CE_i, ED_i)) \cup \{\mathsf{sync}.\checkmark, \mathsf{icpt}.ev \mid ev \in CE_i\} \\ \cup \{\mathsf{link}_{j,k}.dr \mid j, k \in Id, dr \in DR\} \end{array} \right) = \emptyset$.

Conditions (b) and (c) in Definition 1 restrict communication with local policies and enforcers to the intended interfaces. Condition (d) ensures that monitored programs do not interfere with the communication events of service automata. For a given instantiation, we use CE to denote the set of all events of the overall system that are somewhere security-critical, i.e. $CE := \bigcup_{i \in Id} CE_i$. Analogously, we use $ED := \bigcup_{i \in Id} ED_i$ for the set of all enforcement decisions.

4 Instantiation of Service Automata

When using service automata in order to reliably enforce security, the generic model must be adequately instantiated. In particular, the policy and enforcer component have been left underspecified in the previous section. We now give example instantiations of these two components. Which instantiation should be chosen in a concrete scenario depends on the actual context. A complete instantiation for a concrete application scenario will be presented in Section 5.

4.1 Instantiation of Enforcement

It is the responsibility of the enforcer to implement decisions of the coordinator. Several solutions to dynamically react to attempted policy violations have been proposed in the literature on generic runtime monitoring (e.g. [14,10]). In this section, we show how these countermeasures can be implemented in our framework of service automata. Historically, the first proposal was to just stop the program execution upon the occurrence of a policy-violating event [14]. Several other possibilities have been proposed since. We illustrate how one can instantiate the enforcer to realize such approaches. To unify our exposition, we define the set of enforcement decisions to be composed of a critical action and an enforcement action, i.e. $ED_i := \beta \times EA$, where β models the set of security-critical actions while the set EA will be instantiated for each enforcer differently.

Our first example is the terminator TERM that stops the program, when a policy-violating event occurs. We define the set of enforcement actions for the terminator as $EA = \{perm, term\}$ and specify this enforcer as follows:

$$\begin{aligned}\mathsf{TERM}(\beta) \stackrel{\text{def}}{=} \quad & \mathsf{enf}?(ev, a){:}\ \beta{\times}\{perm\} \rightarrow ev \rightarrow \mathsf{sync}!\checkmark \rightarrow \mathsf{TERM}(\beta) \\ \Box\ & \mathsf{enf}?(ev, a){:}\ \beta{\times}\{term\} \rightarrow \mathsf{STOP}\end{aligned}$$

If the enforcement action is *perm* (for permit), the enforcer executes the permitted event and unblocks the interceptor (via sync). If the enforcement action is *term* (for terminate), the terminator halts the program by not unblocking the interceptor. This enforcer safely prevents the current as well as future policy violations.

Another countermeasure known from the literature is suppression [10]. The suppressor SUPP simply skips a policy-violating event without halting the monitored program. We define the set of enforcement actions for the suppressor as $EA = \{perm, supp\}$ and specify this enforcer as follows:

$$\begin{aligned}\mathsf{SUPP}(\beta) \stackrel{\text{def}}{=} \quad & \mathsf{enf}?(ev, a){:}\ \beta{\times}\{perm\} \rightarrow ev \rightarrow \mathsf{sync}!\checkmark \rightarrow \mathsf{SUPP}(\beta) \\ \Box\ & \mathsf{enf}?(ev, a){:}\ \beta{\times}\{supp\} \rightarrow \mathsf{sync}!\checkmark \rightarrow \mathsf{SUPP}(\beta)\end{aligned}$$

Again, if the enforcement action is *perm*, the suppressor executes the permitted event and then unblocks the interceptor. If the enforcement action is *supp* (for suppress), the enforcer skips the critical event and unblocks the interceptor (via sync). In comparison to TERM, SUPP is less rigorous because after suppressing a policy-violating event, it allows the monitored program to continue.

Another standard countermeasure is replacement. The replacer substitutes the policy-violating event by a sequence of events and can also halt the program. We define the set of enforcement actions for the replacer by $EA = (\beta \cup \{stop\})^*$ and instantiate the replacer as follows:

$$\begin{aligned}
\mathsf{REPLACE}(\beta) &\stackrel{\text{def}}{=} \mathsf{enf?}(ev, \sigma)\colon \beta \times EA \rightarrow \mathsf{REPL}(\beta, \sigma) \\
\mathsf{REPL}(\beta, \langle\,\rangle) &\stackrel{\text{def}}{=} \mathsf{sync!}\checkmark \rightarrow \mathsf{REPLACE}(\beta) \\
\mathsf{REPL}(\beta, \langle stop \rangle.\sigma) &\stackrel{\text{def}}{=} \mathsf{STOP} \\
\mathsf{REPL}(\beta, \langle e \rangle.\sigma) &\stackrel{\text{def}}{=} e \rightarrow \mathsf{REPL}(\beta, \sigma)
\end{aligned}$$

This enforcer simply receives an event sequence σ as enforcement action. The policy-violating event is then replaced by this event sequence by recursively calling the process REPL. If the sequence is empty (i.e. $\langle\,\rangle$), the process unblocks the interceptor (via sync). If the sequence starts with the special event *stop*, the replacer terminates and halts the interceptor by not synchronizing.

The replacer is a powerful enforcer. It subsumes all enforcers above as it allows to execute a permissible event *ev* (by replacing it with itself), to halt a malicious program, and to skip a policy-violating event (by replacing *ev* with $\langle\rangle$). For example, if the result of a database query leaks personal information like 'Alice earns 6000 Euros', the query result could be replaced by an error message: 'You are not authorized for personal information'.

All example instantiations above correspond to well-known enforcers. Security automata as defined in [14] use termination as the only enforcement action. Effectively, a security automaton is a service automaton without communication capabilities and with *term* and *perm* as the only enforcement actions. Edit automata have been introduced in [10]. They allow to terminate a program (called truncation in [10]), suppress program events and insert action sequences. Effectively, an edit automaton is a service automaton without any communication capabilities and with REPLACE as enforcer. Naturally, one has to choose an enforcer that is adequate for a given application scenario. If none of the three predefined enforcers is suitable, one still has the possibility to invent an application-specific enforcer when defining the instantiation. That is, the enforcers defined so far should be seen as the nucleus of a growing library.

4.2 Instantiation of the Local Security Policies

In order to reduce conceptual complexity, a local security policy can be specified in a modular manner. This allows one to address complementary aspects such as decision-making, delegation of decisions and routing separately. Moreover, a policy or policy component can also be defined in a stepwise manner, as we illustrate here at the example of a routing component. Our generic specification of a routing component will be specialized for a concrete setting in Section 5.

The role of such a routing component within a local security policy is visualized in Figure 3, which also depicts the communication channels of a policy. Note, that the specification of the decision-making and delegation component (which is left underspecified here) can again be defined in a modular fashion.

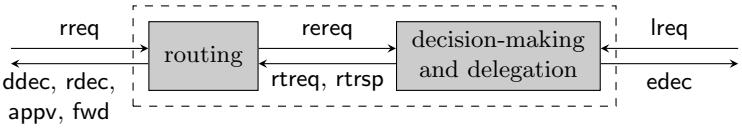

Fig. 3. Composition of a local policy from two components

For simplicity, we assume in our example a *static routing policy* that uses a fixed route between any two service automata. The function $nxt(i,k)$ determines the next node on the route from service automaton i to the final destination k. The set DR of delegation requests and responses is defined as $DR := Id \times CE \cup Id \times ED$ where the first component denotes the final destination of the request or response and the second component is either the critical event (in CE) for which a decision is requested or a decision (in ED) for a previous request. The channels rereq, rtreq and rtrsp are used for the communication between the routing and the delegation and decision-making policy component. We specify this generic router as follows:

$$\begin{aligned}\mathsf{SRP}_i \stackrel{\text{def}}{=}_{\alpha} \quad & \mathsf{rreq}?(k,x)\colon \{(k',x') \in DR \mid k' \neq i\} \to \mathsf{fwd}!(nxt(i,k),(k,x)) \to \mathsf{SRP}_i \\ & \Box\, \mathsf{rreq}?(k,ev)\colon \{i\}\times CE \to \mathsf{rereq}!\,ev \to \mathsf{SRP}_i \\ & \Box\, \mathsf{rreq}?(k,ed)\colon \{i\}\times ED \to \mathsf{appv}!\,ed \to \mathsf{SRP}_i \\ & \Box\, \mathsf{rtreq}?(k,ev)\colon Id\times CE \to \mathsf{ddec}!(nxt(i,k),(k,ev)) \to \mathsf{SRP}_i \\ & \Box\, \mathsf{rtrsp}?(k,ed)\colon Id\times ED \to \mathsf{rdec}!(nxt(i,k),(k,ed)) \to \mathsf{SRP}_i\end{aligned}$$

This router accepts a delegation request (k, ev) or response (k, ed) from the coordinator (via rreq). If this request or response has not reached its final destination, the coordinator is informed (via fwd) that it should forward the request or response to the next node on the path to k. Otherwise, if it is a request, then it is passed on to the decision-making and delegation component (via rereq), and if it is a response, then an approval for this decision is sent to the coordinator (via appv). This router also accepts delegation requests (via rtreq) and responses (via rtrsp) from the delegation and decision-making component and forwards them to the coordinator together with the identifier of the next node on the route to the final destination (via ddec or rdec). The alphabet α of this router is $\alpha := \{\mathsf{rreq}.dr, \mathsf{fwd}.(k,dr), \mathsf{ddec}.(k,dr), \mathsf{rdec}.(k,dr), \mathsf{appv}.ed, \mathsf{rereq}.ev, \mathsf{rtreq}.(k,ev), \mathsf{rtrsp}.(k,ed') \mid k \in Id \setminus \{i\}, ev \in CE, ed \in ED_i, ed' \in ED, dr \in DR\}$.

This generic specification of the routing component can be instantiated for a concrete application scenario by defining the function nxt based on the communication structure that is available in the given distributed system.

5 Stepwise Instantiation in an Application Scenario

In this section, we demonstrate the stepwise instantiation of service automata for an example scenario in which a Chinese Wall security policy shall be enforced.

The scenario. We consider a distributed service (depicted in Figure 4) in which a set of service providers ($SP = \{sp_1, \ldots, sp_m\}$) offers services to users ($U =$

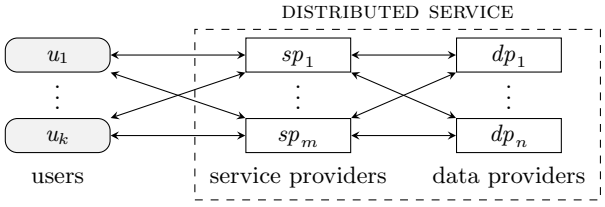

Fig. 4. Application Scenario

$\{u_1, \ldots, u_k\}$). A user places a query to a service provider who then requests the information that is necessary for computing this query from the data providers ($DP = \{dp_1, \ldots, dp_n\}$). The data providers return the requested data objects to the service provider. After receiving all necessary objects from the data providers, the service provider computes the query result and sends it to the user.

We model the responses of a data provider dp to a request by events of the form $ev = (u, \{o\}, dp, sp)$, where u denotes the user on whose behalf the request was placed, $\{o\}$ is a singleton set containing the requested object and sp is the service provider to whom the response is addressed. Moreover, we model the replies of a service provider sp to queries by events of the form $ev' = (u, O', sp)$, where u is the user who has placed the query and $O' \subseteq O$ denotes the set of objects addressed by this query. We assume that for each event ev and ev', there is a unique event $dummy_{ev}$ that constitutes a legitimate response that does not reveal any information about objects (e.g. an error message).

The security requirement. In our scenario, data objects may belong to competing companies. We say, that such objects are *in conflict* and capture this by an irreflexive and symmetric conflict of interest relation $COI \subseteq O \times O$ on objects. Any query result that a service provider delivers to a user must neither depend on conflicting objects nor on objects that are in conflict with objects used for computing results requested earlier by the same user.

Towards an Instantiation. In this section, we show how coordinated enforcement can be used to enforce the Chinese Wall policy in the above example. We encapsulate all service providers and data providers by service automata. A function *resp* maps each policy-relevant event that is performed by the monitored data or service provider to the identifier of the service automaton that is *responsible* to decide on the event. For events that address an object residing on a data provider which also stores all conflicting objects, the service automaton of this data provider is responsible. If a service provider has exclusive access to all objects in given COI sets, this service provider is responsible for events modeling queries that involve only objects from those COI sets. Finally, if a user only uses a single service provider, then the service automaton that encapsulates this service provider is responsible for all events modeling queries by this user. For events accessing objects of COI sets which do not fall under the previous cases we are free to choose an arbitrary service automaton to be responsible. Note, that if

only the first three cases are present, the Chinese Wall policy could already be enforced by local monitors. However, this condition is somewhat restrictive and it is likely, that it is not satisfied. In this case, coordinated enforcement is required.

Instantiation. We instantiate the set Id by $Id := \{dp_1, \ldots, dp_n, sp_1, \ldots, sp_m\}$ assuming that the monitored programs of the service and data providers are represented by process expressions PROV_i for $i \in Id$. The set of critical events CE_i for each $i \in Id$ is a subset of $\alpha(\mathsf{PROV}_i)$ that is defined as follows: In case i is a data provider, the critical events CE_i contain all responses to access requests. These events are either of the form $ev = (u, \{o\}, i, sp)$ or $dummy_{ev}$. In case i is a service provider, the critical events CE_i contain all replies to former user queries. These events are either of the form $ev' = (u, O', i)$ or $dummy_{ev'}$. Note, that $CE_i \cap CE_j = \emptyset$ holds for $i \neq j$. We define $obj(ev) \subseteq O$ to denote the set of objects that are contained in ev and $u(ev) \in U$ to denote the user contained in ev. If $ev \in CE_i$ for some i, then we define $id(ev) = i$ (otherwise id is undefined). We lift the conflicts on objects to conflicts on critical events: two events are in conflict, denoted $ev \otimes ev'$, iff they are not dummy events and they access conflicting objects on behalf of the same user, i.e., $u(ev) = u(ev')$ and there exist $o_1 \in obj(ev)$, $o_2 \in obj(ev')$ with $(o_1, o_2) \in COI$.

We now instantiate the local policy and the enforcer. The policy is the parallel composition of a decision-making component, a delegation component and a routing component. The decision-making component (to be specified below) communicates with the delegation component (to be specified below) via lereq, and with the routing component via rereq (to be specified below) and rtrsp. The delegation component communicates with the routing component via rtreq.

Instantiating decision-making. Whether a critical event is permissible, depends on critical events that have previously been performed on behalf of the same user. Hence, the decision-making component of the local policy collects all previously performed events in its state $q \in 2^{CE}$. In a state q, an event ev shall be rejected and replaced by a dummy event if it is in conflict with any previously accessed event, i.e., $ev \in conf(q)$ for $conf(q) = \{ev \in CE \mid \exists ev' \in q \cup \{ev\}.(ev \otimes ev')\}$. We instantiate the enforcement decisions by $ED_i := CE_i \times CE_i$.

$$
\begin{aligned}
\mathsf{DEC}_i(q) \stackrel{\text{def}}{=}_{\alpha}\ & \mathsf{lereq}?\,ev\colon\ CE_i \cap conf(q) \rightarrow \mathsf{edec}!(ev, dummy_{ev}) \rightarrow \mathsf{DEC}_i(q) \\
\Box\ & \mathsf{lereq}?\,ev\colon\ CE_i \setminus conf(q) \rightarrow \mathsf{edec}!(ev, ev) \rightarrow \mathsf{DEC}_i(q \cup \{ev\}) \\
\Box\ & \mathsf{rereq}?\,ev\colon\ conf(q) \rightarrow \mathsf{rtrsp}!(id(ev), (ev, dummy_{ev})) \rightarrow \mathsf{DEC}_i(q) \\
\Box\ & \mathsf{rereq}?\,ev\colon\ CE \setminus conf(q) \rightarrow \mathsf{rtrsp}!(id(ev), (ev, ev)) \rightarrow \mathsf{DEC}_i(q \cup \{ev\})
\end{aligned}
$$

The process accepts local and remote decision requests (via lereq and rereq). In case of a local request, it sends its decision to the coordinator (via edec). In case of a remote request, this enforcer sends its decision to the routing component (via rtrsp). If the enforcer permits the event, it updates its state q. If it rejects the event, it replaces the policy-violating event by a dummy event. The process engages in all communication on the used channels with alphabet $\alpha = \{\mathsf{lereq}.ev, \mathsf{edec}.ed \mid ev \in CE_i, ed \in ED_i\} \cup \{\mathsf{rereq}.ev, \mathsf{rtrsp}.(k, ed) \mid ev \in CE, ed \in ED, k \in Id\}$.

Instantiating delegation. The delegation component identifies the responsible service automaton for deciding on a critical event. We define the function *resp* such that it determines the same responsible automaton for any two conflicting events, i.e. for all ev, ev' with $ev \otimes ev'$ we have $resp(ev) = resp(ev')$. Based on *resp*, the delegation component determines, whether an event can be decided locally or needs to be delegated.

$$\begin{aligned}\mathsf{DEL}_i \stackrel{\text{def}}{=}_{\alpha} \quad & \mathsf{lreq}?ev\colon \{ev' \in CE_i \mid i = resp(ev')\} \to \mathsf{lereq}!ev \to \mathsf{DEL}_i \\ \Box\, & \mathsf{lreq}?ev\colon \{ev' \in CE_i \mid i \neq resp(ev')\} \to \mathsf{rtreq}!(resp(ev), ev) \to \mathsf{DEL}_i\end{aligned}$$

The process accepts a local event (via lreq). If the local service automaton is responsible ($i = resp(ev)$), the policy requests local decision-making (via lereq). Otherwise, it instructs the routing process (via rtreq) to determine the route to $resp(ev)$. In the process definition, $\alpha = \{\mathsf{lreq}.ev, \mathsf{lereq}.ev \mid ev \in CE_i\} \cup \{\mathsf{rtreq}.(k, ev) \mid ev \in CE, k \in Id \setminus \{i\}\}$ ensures that DEL_i participates in all communication on the used channels.

Instantiating routing. In our example, the communication structure between service automata is a fully connected graph that does not change over time. We use the static routing subcomponent SRP_i defined in Section 4.2 and concretize the function $nxt(i, k)$ such that it returns destination k as the next node.

Instantiating the enforcement. In our scenario, policy-violating events are replaced by the corresponding dummy events determined by the decision-making. This is implemented by the enforcer $\mathsf{REPLACE}(CE_i)$ of Section 4.1 with replacement sequences EA concretized by CE_i. How critical events are replaced is specified in the definition of $\mathsf{DEC}_i(q)$ above.

Service automata framework. The instantiation of the local policy is the parallel composition $\mathsf{POL}_i \stackrel{\text{def}}{=} (\mathsf{DEL}_i \parallel \mathsf{DEC}_i(\emptyset) \parallel \mathsf{SRP}_i) \setminus H_i^{\text{pol}}$, which hides the internal communication with the set

$$H_i^{\text{pol}} := \left\{ \begin{array}{l} \mathsf{lereq}.ev, \mathsf{rereq}.ev, \\ \mathsf{rtreq}.(k, ev), \mathsf{rtrsp}.(k, ed) \end{array} \middle| \begin{array}{l} ev \in CE \\ ed \in ED, k \in Id \setminus \{i\} \end{array} \right\}.$$

Then the controlled system is the instantiated service automata framework

$$\mathsf{SYSTEM} \stackrel{\text{def}}{=} \parallel_{i \in Id} \mathsf{SA}_i(\mathsf{PROV}_i, CE_i, \mathsf{POL}_i, ED_i, \mathsf{REPLACE}(CE_i))$$

Soundness of the Enforcement. We formalize the Chinese Wall security requirement of our scenario by the following definition:

Definition 2. *For sequences tr of events, we define the predicate ChW by*

$$ChW(tr) := \neg\exists ev_1, ev_2 \in CE.(\, ev_1 \lhd tr \,\wedge\, ev_2 \lhd tr \,\wedge\, ev_1 \otimes ev_2).$$

That is, in a single trace, no conflicting accesses may occur. With this definition, the controlled system soundly enforces the Chinese Wall policy.

Theorem 1. SYSTEM sat *ChW*.

Proof (sketch[1]*).* Suppose, Theorem 1 does not hold. Then there exists a system trace tr that contains two conflicting events ev_1 and ev_2 (with $ev_1 \otimes ev_2$). The instantiation of the enforcer ensures that an event is performed only after a corresponding permission decision has been received. We can show, that the permission decisions for ev_1 and ev_2 must have been made by their respective responsible nodes. Since ev_1 and ev_2 are in conflict, they have the same responsible node (by the definition of function *resp*). This node must hence have permitted both events ev_1 and ev_2. However, this contradicts the instantiation of the decision-making component together with the definition of function *conf*. Hence, tr cannot contain both conflicting events ev_1 and ev_2 and Theorem 1 holds. □

6 Related Work

Generic security monitoring was pioneered by Schneider's framework of security automata [14]. While security automata and edit automata [10] were designed for securing individual programs, our service automata framework can secure distributed systems in a decentralized, coordinated fashion.

Another framework for monitoring policies in distributed, service-based systems is proposed on a conceptual level in [6]. However, unlike our approach, this framework is restricted to monitoring without enforcement and, moreover, local monitors can only communicate via a central unit.

A similar restriction appears in the framework for coordinated decision-making proposed in [5]. Local monitors are composed from three components: a policy information point (PIP), a policy decision point (PDP), and a policy enforcement point (PEP). While a local monitor can make enforcement decisions by itself, this capability is limited because PDPs are stateless. In order to remedy this deficiency, local monitors can communicate with special coordination objects that are stateful. However, local monitors cannot communicate directly with each other and coordination objects cannot communicate among each other either. This limits the possibilities for coordinated, decentralized enforcement.

In [15], an approach to decentralized monitoring is proposed and formalized in a temporal logic. In this approach, a monitor piggy-backs information about its local state onto regular messages of the monitored program. In contrast to our approach, no additional messages need to be introduced. The drawback is that monitors cannot trigger communication themselves and, hence, have to rely on information about other nodes that might be outdated. This makes a sound enforcement of global security aspects impossible if these depend on up-to-date information about remote nodes (such as Chinese Wall policies).

The law-governed interaction framework [12] performs monitoring and enforcement based on the interception and alteration of messages exchanged between nodes of the given distributed system. In contrast, service automata can observe individual computation steps of a monitored program, which results in more

[1] The full formal proof is available on the authors' website.

fine-grained information for making enforcement decisions. While the framework in [12] has been implemented, it lacks a formal model or soundness result.

In [11], an approach to synthesize decentralized monitors for enforcing policies in distributed systems is described. However, the synthesized controllers cannot communicate with each other. Hence, global security requirements such as, e.g, Chinese Wall policies cannot be enforced (as already pointed out in [11]). The same limitation applies to the distributed usage control approach proposed in [1].

The process algebra that we employ in this paper, i.e. CSP [9] has already been used to formalize generic security monitors in [2] and, in combination with Object Z, in [4]. Like in our approach, a monitor synchronizes with a program on all security-critical actions. However, monitors lack communication capabilities and, moreover, termination is the only countermeasure against policy violations that is supported. In contrast to this, our service automata can coordinate their actions to enforce global security requirements, and they support a wider range of countermeasures including termination, suppression, replacement, and others.

The application of run-time monitoring for usage control is gaining popularity. In [13], a translation from high-level usage control policies to low-level policies is proposed that respects monitoring capabilities by distinguishing controllable, observable and non-observable aspects. The article also proposes an enforcement architecture for data providers while the enforcement architecture in [16] focuses on data consumers and relies on a translation of usage control policies into low-level access control policies. Formal semantics for some usage control policies exist (see, e.g., [8]), but a satisfactory formal model for the enforcement of distributed usage control is yet lacking. Service automata might be able to fill this gap.

7 Conclusion

In this article, we proposed service automata as a framework for enforcing security requirements in distributed systems at runtime. We developed a generic security model in CSP that can be instantiated in a stepwise and modular fashion. As an example, we presented a specialization for an application scenario and used the resulting formal model to prove that Chinese Wall policies are soundly enforced. The formal security model could also be used to prove that an implementation of service automata is sound, but this is outside the scope of the current article.

We are confident that service automata provide a very suitable basis for enforcing security in distributed systems, including aspects of access control and usage control. In the future, we plan to explore this spectrum further, provide an efficient implementation of the service automata framework and use our formal security model of service automata to prove the soundness of the implementation.

Acknowledgements. We thank Sarah Ereth for her feedback and the anonymous reviewers for their constructive comments. This work was partially funded by CASED (www.cased.de) and by the DFG (German research foundation) under the project FM-SecEng in the Computer Science Action Program (MA 3326/1-3).

References

1. Aziz, B., Arenas, A., Martinelli, F., Matteucci, I., Mori, P.: Controlling Usage in Business Process Workflows through Fine-Grained Security Policies. In: Furnell, S., Katsikas, S.K., Lioy, A. (eds.) 5th International Conference on Trust, Privacy and Security in Digital Business. pp. 100–117. LNCS 5185, Springer (2008)
2. Basin, D.A., Burri, S.J., Karjoth, G.: Dynamic Enforcement of Abstract Separation of Duty Constraints. In: Backes, M., Ning, P. (eds.) 14th European Symposium on Research in Computer Security. pp. 250–267. LNCS 5789, Springer (2009)
3. Basin, D.A., Klaedtke, F., Müller, S.: Policy Monitoring in First-Order Temporal Logic. In: Touili, T., Cook, B., Jackson, P. (eds.) 22nd International Conference on Computer Aided Verification. pp. 1–18. LNCS 6174, Springer (2010)
4. Basin, D.A., Olderog, E.R., Sevinç, P.E.: Specifying and analyzing security automata using CSP-OZ. In: ACM Symposium on Information, Computer and Communications Security. pp. 70–81. ACM (2007)
5. Chadwick, D.W., Su, L., Otenko, A., Laborde, R.: Coordination between Distributed PDPs. In: 7th IEEE International Workshop on Policies for Distributed Systems and Networks. pp. 163–172. IEEE Computer Society (2006)
6. Comuzzi, M., Spanoudakis, G.: A Framework for Hierarchical and Recursive Monitoring of Service Based Systems. In: 4th International Conference on Internet and Web Applications and Services. pp. 383–388. IEEE Computer Society (2009)
7. Erlingsson, U., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: 2nd New Security Paradigms Workshop. pp. 87–95. ACM (2000)
8. Hilty, M., Pretschner, A., Basin, D.A., Schaefer, C., Walter, T.: A Policy Language for Distributed Usage Control. In: 12th European Symposium on Research in Computer Security. pp. 531–546 (2007)
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc. (1985)
10. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security 4(1–2), 2–16 (2005)
11. Martinelli, F., Matteucci, I.: Synthesis of Local Controller Programs for Enforcing Global Security Properties. In: 3rd International Conference on Availability, Reliability and Security. pp. 1120–1127. IEEE Computer Society (2008)
12. Minsky, N.H.: The Imposition of Protocols Over Open Distributed Systems. IEEE Transactions on Software Engineering 17(2), 183–195 (1991)
13. Pretschner, A., Hilty, M., Basin, D.: Distributed Usage Control. Communications of the ACM 49(9), 39–44 (2006)
14. Schneider, F.B.: Enforceable Security Policies. Transactions on Information and System Security 3(1), 30–50 (2000)
15. Sen, K., Vardhan, A., Agha, G., Roşu, G.: Efficient Decentralized Monitoring of Safety in Distributed Systems. In: 26th International Conference on Software Engineering. pp. 418–427. IEEE Computer Society (2004)
16. Zhang, X., Seifert, J.P., Sandhu, R.: Security Enforcement Model for Distributed Usage Control. In: 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing. pp. 10–18. IEEE Computer Society (2008)