

# Service Automata

Java Implementation Manual

Richard Gay

May 30, 2013

# 1 Introduction

Service Automata [GMS12] are a dynamic enforcement mechanism. Service Automata particularly address dynamic enforcement on distributed programs. For an introduction to the concept of Service Automata and a formal model of Service Automata in the process algebra CSP [Hoa85], we refer the reader to [GMS12].

This document describes an implementation of Service Automata in Java and how the implementation can be instantiated for enforcing a security requirement on a distributed program.

## 1.1 Terminology

This manual makes use of several terms, for which the following list gives an overview.

target program: A target program in the context of Service Automata is a distributed program on which security requirements are supposed to be enforced by using the Service Automata implementation.

agent: An agent is a non-distributed entity of a (distributed) target program.

Service Automaton: A Service Automaton is a non-distributed entity of the Service Automata framework. A Service Automaton can be considered to “encapsulate” one agent of a target program. That is, it ensures that the operations that the agent attempts to perform are mediated in accordance with the security requirements.

## 2 Framework Architecture

Service Automata have a modular architecture. Figure 2.1 shows this architecture.

interceptor: fixed code, which first uses the event factory to turn a concrete operation (join point) into an abstracted critical event representation and second sends this event to the coordinator

event factory: a parametric component that has one factory method for each pointcut that returns an abstracted critical event for every join point matching the pointcut (see Section 3.4)

CE: critical event; an object of a data type

coordinator: a component that manages the communication between the components

local policy: a parametric component that makes local decisions and is able to delegate decisions (see Section 3.6)

DR: delegation request/response data type, used to determine the message format for the communication between Service Automata

ED: enforcement decision data type (see Section 3.5)

enforcer executor: fixed code, which receives enforcement decisions, uses the enforcer factory to turn a decision into a concrete enforcer, and runs the enforcer

enforcer: an enforcer is a component that can be executed and whose execution realizes enforcement decisions such as termination (see Section 3.5)

enforcer factory: an enforcer factory is a parametric component providing a factory method that produces enforcer objects for enforcement decision objects

unit starter: this fixed component bootstraps a Service Automaton by starting the instrumented Java agent as well as the program containing the coordinator and local policy components

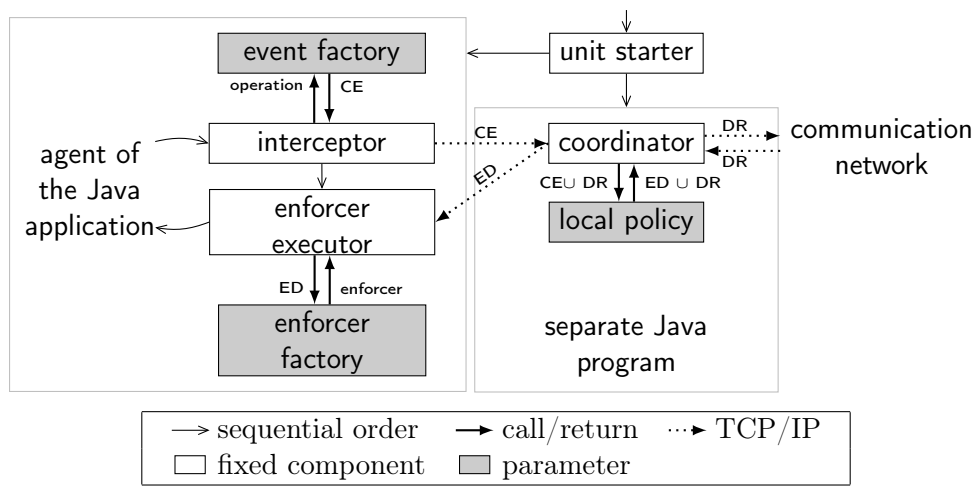


Figure 2.1: Architecture of a Service Automaton in Java

## 3 Instantiation of the Framework

The implementation of Service Automata is parametric. The parameters determine how the enforcement takes place and on which target program it takes place.

The remainder of this section gives a list of steps, along which one can instantiate the parameters for a concrete application scenario. Note that this is not meant to be the only possible approach.

### 3.1 Determining the Agents of the Target Program

The concept of Service Automata stipulates encapsulating agents of the target program with Service Automata. When instantiating the Service Automata implementation, one thus has to determine the agents that are supposed to be subject to dynamic enforcement. This leaves a design choice. In a client-server system, for instance, it might suffice to consider only client agents, to consider only server agents, or to consider both client and server agents.

Which agents are to be chosen may depend on several factors, such as

- which agents perform security-relevant operations,
- which agents are sufficiently controllable such that they can be encapsulated by Service Automata, and/or
- by encapsulating which agents would the enforcement be particularly efficient (e.g., because local decisions can be made more often).

For the following, we assume that a set of agents has been chosen. For each agent, a JAR file must be available. This JAR file will be used in Section 3.2.

### 3.2 Configuring the Instantiation

An instantiation of the Service Automata implementation is determined by several parts. The parts are specified in the form of a configuration file. Listing 3.1 shows the syntax and the configuration file by example.

```

1  cfg.nodes = agent1, agent2, agent3, agent4
2  cfg.destdir = /tmp/target/
3
4  node.type = Java
5  node.target = path/to/agent.jar
6  node.target-javavm = /usr/bin/java
7  node.pointcuts = path/to/pointcuts.txt
8  node.policy = fqcn.of.MyLocalPolicy
9  node.event-factory = fqcn.of.MyEventFactory
10 node.enforcer-factory= fqcn.of.MyEnforcerFactory
11 node.inline-classpath= path/to/inlined-classes.jar:log4j:jansi
12 node.policy-classpath= path/to/policy-classes.jar:log4j
13 node.loglevel = WARN
14 node.ext-port = 1901
15 node.cor-port = 1902
16 node.enf-port = 1903
17 node.cor-host = localhost
18 node.enf-host = localhost
19
20 agent1.ext-host = 192.168.100.1
21 agent2.ext-host = 192.168.100.2
22 agent3.ext-host = 192.168.100.3
23 agent4.ext-host = 192.168.100.4

```

Listing 3.1: Configuration file for an instantiation

Configuration files use the syntax of Java’s property files <sup>1</sup>. The format is line-based and assigns values to keys. Key names are structured into categories with separating dots.

The key in Line 1 configures the nodes of the distributed encapsulated system, which each consist of an agent of the target program as well as a Service Automaton. The value in this line is a comma-separated list of unique names for these nodes. Line 2 configures the directory in which the result of the encapsulation shall be written. For each node, one subdirectory is created.

The keys in Lines 4 to 18 share the prefix “node”. These lines determine the configuration that is the same for all nodes. On the other hand, the keys in Lines 20 to 23 are prefixed with a node’s name and determine configuration that applies to the respective node only. The following describes the purpose of these configuration elements:

---

<sup>1</sup>see, e.g., <http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html>

type: type of the agent; currently supports only “Java”; the following configuration elements assume agents of type Java

target: path to the JAR file containing the code of the agent; if the path is relative, then it is considered relative to the configuration file

tagret-javavm: path to the Java VM on the machines where the respective agent is supposed to be executed

pointcuts: list of pointcuts that determine security-relevant operations and, as such, the points in the code at which the interceptor and enforcer are inlined (see Section 3.3 for details)

policy: fully qualified name of the local policy class (see Section 3.6 for details)

event-factory: fully qualified name of the event factory class (see Section 3.4)

enforcer-factory: fully qualified name of the enforcer factory class (see Section 3.5)

inline-classpath: colon-separated list of classpath entries that are added to the JAR file of the target agent; the list must include the JAR file(s) that contain the critical event classes, the critical event factory, enforcement decision classes, enforcer classes, the enforcer factory

policy-classpath: colon-separated list of classpath entries that are added to the Java program containing the local policy; the list must include the JAR file(s) that contain the local policy, the critical event classes, and the enforcement decision classes

loglevel: log level of log4j<sup>2</sup>

ext-host: host name, with which the node’s Service Automaton can be addressed by other Service Automata

ext-port: port number, with which the node’s Service Automaton can be addressed by other Service Automata

cor-host: host name, with which the Service Automaton’s interceptor component can connect to the coordinator; can possibly be set to “localhost” often, given that both components run on the same machine

cor-port: port number, with which the Service Automaton’s interceptor component can connect to the coordinator

enf-host: host name, with which the Service Automaton’s coordinator component can connect to the enforcer; can possibly be set to “localhost” often, given that both components run on the same machine

enf-port: port number, with which the Service Automaton’s coordinator component can connect to the enforcer

The remaining sections of this chapter describe how the values for the “point-

---

<sup>2</sup>see, e.g., <http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html>

```

PointcutSpec      ::= ((PointcutDecl | Import) ';'*)
PointcutDecl      ::= PointcutMods 'pointcut' Identifier
                        Formals ':' PointcutExpr
PointcutMods      ::= ('public' | 'private' | 'protected')*
Formals           ::= '(' ParamList? ('>' FullyQualifiedName )? ')',
PointcutExpr      ::= PointcutPrim
                        | '!' PointcutExpr | '(' PointcutExpr ')',
                        | PointcutExpr '&&' PointcutExpr
                        | PointcutExpr '||' PointcutExpr
PointcutPrim      ::= Identifier '(' JavaExpression ')',
ParamList         ::= FullyQualifiedName Identifier (',' ParamList)*
Import            ::= 'import' FullyQualifiedName
FullyQualifiedName ::= Identifier ('.' Identifier)*
Identifier        ::= ['a'-'z','A'-'Z'] ( ['a'-'z','A'-'Z','0'-'9'] )*

```

Figure 3.1: Language for specifying security-relevant operations

cuts” (Section 3.3), “policy” (Section 3.6), “event-factory” (Section 3.4), “enforcer-factory” (Section 3.5) keys can be configured.

### 3.3 Declaring Security-Relevant Operations

This step consists of two parts: first, identifying security-relevant operations of a given target program and, second, specifying these operations in a form that is understandable by the Service Automata implementation.

Identifying security-relevant operations leaves a design space, similar to the choice of the agents of the target program. Security-relevant operations may be chosen at a low program level (e.g., single instructions) or at a more coarse-grained level (e.g., whole method bodies). Concerning methods, one can in principle choose API methods as well as program-defined methods.

Currently, the Service Automata implementation has been designed with method calls as the security-relevant operations as the focus. In the following, we assume that for each agent the set of security-relevant method calls has been chosen. The next step is to specify these operations.

Security-relevant operations are specified as a **PointcutSpec** of Figure 3.1. The language essentially is a list of pointcut specifications from AspectJ [KHH<sup>+</sup>01]. For this, any **Identifier** of **PointcutPrim** must be an identifier supported by AspectJ, such as “call”.

The language extends the AspectJ pointcut language by a return type, the **FullyQualifiedName** in the **Formals** non-terminal. If this is omitted, **void** is



assumed as the return type.

### 3.4 Implementing Event Abstractions

The Service Automata implementation at runtime intercepts all attempts by an agent to perform a security-relevant method that was specified as part of Section 3.3. Whenever an agent attempts to perform a security-relevant method call, then right before this attempt, the object on which the method is supposed to be called as well as all parameter values are known. All this information could be used for making a decision about the operation.

Since often not all information available at a method call are actually required for making a decision, the Service Automata implementation supports event abstractions. When instantiating the Service Automata implementation for a concrete setting, one therefore has to perform the following steps for each agent:

- (a) Identify relevant information for making decisions about events.

This step heavily depends on the security requirements to be enforced. If, for instance, the security requirements speak about users' activity, then user names might be relevant information that should be present in abstracted events.

- (b) Define Java data types (by implementing the `CriticalEvent` interface) for the events, such that all the information can be captured by the data types.
- (c) Implement a factory class that constructs critical event objects from concrete method calls.

For each security-relevant operation of the agent, specified in Section 3.3, there must be a corresponding factory method of the same name (the `Identifier` of `PointcutDecl`) in the factory class. This method must have the same parameter list as the pointcut (`ParamList` in `Formals`) and must return a `CriticalEvent` object.

### 3.5 Implementing Countermeasures

The goal of a dynamic enforcement mechanism is to impose appropriate countermeasures against impending violations of security requirements by the target program. Operations that would not constitute a violation should be permitted. We first look at how countermeasures are specified and implemented and afterwards, in Section 3.6, describe how countermeasures are decided.

The first step towards countermeasures is to define possible decisions that shall lead to countermeasures. For this, an instantiation of the Service Automata implementation derives one or multiple Java classes from the `EnforcementDecision` interface. These classes are used for storing decisions (such as “permit” or “terminate”) as well as possible auxiliary information that is needed for carrying out the decision (such as an informative message to be displayed to a user). Ideally, the decisions are technically independent from particularities of the target program’s agent.

The second step is to define how decisions are translated to concrete executable countermeasures. For this, an instantiation must provide an implementation of the `EnforcerFactory` interface, implementing

- **public static** `Enforcer fromDecision(final EnforcementDecision ed)`
- **public static** `Enforcer fallback(final CriticalEvent ev)`

Both methods return an `Enforcer` object. For this to be possible, an instantiation must derive one or more classes from the `Enforcer` interface, which provides the methods

- **public boolean** `suppress()`, which determines whether the original security-relevant operation shall be performed (**true** return value) or not (**false** return value),
- **public void** `before()`, which is code that shall be executed before the original security-relevant operation is performed or suppressed,
- **public void** `after()`, which is code that shall be executed after the original security-relevant operation is performed or suppressed, and
- **public Object** `getReturnValue(Class c)`, which must return a substitute return value if the security-relevant method has a return value and its execution is suppressed.

### 3.6 Implementing (Coordinated) Deciding

For making decisions, an instantiation must derive a class from the abstract `LocalPolicy` class. For a Service Automaton, deciding as part of a distributed enforcement mechanism involves two aspects:

- (a) deciding for operations that were attempted by the respective agent of the target program that the Service Automaton encapsulates;

- (b) deciding for operations that were attempted by a remote agent, which is encapsulated by another Service Automaton.

For supporting these two cases, a `LocalPolicy` must implement the following two methods

```
public abstract LocalPolicyResponse localRequest(CriticalEvent ev)
    throws IllegalArgumentException
```

```
public abstract LocalPolicyResponse remoteRequest(DelegationReqResp dr)
    throws IllegalArgumentException
```

A `LocalPolicyResponse` can be one of `EnforcementDecision` or `DelegationLocPolReturn`, where the latter comprises a `DelegationReqResp` and the identifier of the remote Service Automaton that serves as the delegate. The two possible types of return values correspond to the two kinds of decisions that a Service Automaton can make upon a request:

- (a) an `EnforcementDecision` that can be turned into a countermeasure as described in Section 3.5;
- (b) a (partial) decision that is to be delegated to a remote Service Automaton.

The latter must be an object of a class that implements the (empty) `DelegationReqResp` interface. That is, when implementing an instantiation that uses coordination, one or more classes implementing this interface must be provided. One may define two classes, for instance, one for the requests (which could contain a `CriticalEvent` as well as `Strings` for source and destination identifiers) and one for the responses to requests (which could contain an `EnforcementDecision` as well as `Strings` for source and destination identifiers).

## 4 Encapsulating Distributed Java Programs

Chapters 2 and 3 describe the parametric architecture of a Service Automaton as well as the instantiation of this architecture. Applying an instantiation to a given distributed Java program is as easy as running

```
java -jar path/to/encaps.jar <config.cfg>
```

where the `encaps.jar` is a tool that is part of the Service Automata implementation.

# Bibliography

- [GMS12] R. Gay, H. Mantel, and B. Sprick. Service Automata. In *Proceedings of the 8th International Workshop on Formal Aspects of Security and Trust (FAST)*, LNCS 7140, pages 148–163. Springer, 2012.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.
- [KHH<sup>+</sup>01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, pages 327–353. Springer, 2001.