CESSA

Compositional Evolution of Secure Services using Aspects

ANR project no. 09-SEGI-002-01

Language definition and aspect support

Extension of the Service Model for Security and Aspects

Abstract. In this deliverable the CESSA aspect model is introduced that comprises a notion of gray-box composition, aspect-aware interfaces, and a formal aspect model for service interactions. Our notions of gray-box model and aspect-aware interfaces can be instantiated in order to accommodate different notions of scope, *e.g.,* security domains, and different classes of security properties. The aspect model for service interactions complete more traditional aspect languages that can be used for service implementations, so that horizontal and vertical service compositions can be handled. Finally, we present a number of examples of how security properties can be managed using this model.

Deliverable No.	D1.3
Task No.	1
Type	Public
Dissemination	Public
Status	Final
Version	2
Date	26 September 2011
Authors	D. Allam, T. Bourdier, R. Douence, H. Grall, J.-C. Royer, M. Südholt (EMNantes), A. Santana De Oliveira (SAP)

Contents

Chapter 1

Introduction

In this deliverable the CESSA aspect model for the evolution of service compositions is presented. Following an analysis of high-level requirements for such a model presented in deliverable D1.2 ([2], chp. 5) and a requirement analysis based on the usage scenarios, this model satisfies two main characteristics. First, the model supports the flexible manipulation of service interactions on the choreography and implementation levels, that is, it supports the evolution of horizontal and vertical service compositions. Second, the different parts of the model are either equipped with a formal semantics and support reasoning over security; in the case of mechanisms that are too expressive for automatic or semi-automatic reasoning over security properties, our aspect model provides means to restrict their use by means of interaction protocols so that properties can be tackled.

Concretely, we introduce and formally define an aspect language for service interactions, as well as notions of gray-box composition and aspect-aware interfaces. Our notions of gray-box model and aspect-aware interfaces can be instantiated in order to accommodate different notions of scope, *e.g.,* security domains, and different classes of security properties. The aspect model for service interactions complete more traditional aspect languages that can be used for service implementations, so that horizontal and vertical service compositions can be handled. Finally, we present a number of examples of how security properties can be managed using this model.

Overview of the aspect model

Because of the above evolution requirements summarized above, our aspect model consists of the following parts:

1. An **aspect language for horizontal service compositions**. This language enables the manipulation of service compositions.

2. An **aspect language for vertical service compositions**. This language enables the manipulation of service implementations in terms of processes and process implementations defined using the formal model introduced in deliverable D1.2 [2].

3. A **model for gray-box composition** that allows arbitrary entities of service definitions and service compositions to be accessed and manipulated.

4. A notion of **aspect-aware interfaces** that defines conditions on base computations and aspects that must be satisfied for aspects to be applicable. They can therefore be used to restrict the applicability of aspects that would invalidate desirable properties, notably security properties, of service compositions.

In this deliverable, we mainly define the aspect language for horizontal service compositions. We do not detail the aspect language for vertical service composition, because well-known aspect models and languages, such as the AspectJ model and languages like AO4BPEL [9] can be used for this purpose.

Furthermore, we define the gray-box composition model and aspect-aware interfaces, but only at a high level of abstraction. Both of these parts strongly depend on which base systems they are applied to and, for the latter, which class of properties should be supported. The applications and the infrastructures they are to be applied to will be fixed, however, only in the following phase of the CESSA project, mainly based on requirements by the industrial partners. We therefore provide a conceptual model for gray-box composition and a notion of aspect interfaces that is parameterized by the class of properties. Both of these abstract models will be instantiated in the following phase of the CESSA project.

This deliverable is structured as follows. The next chapter introduces our formal model for Service-Oriented Computing and its type system. Chapter 3 presents our aspect languages and describes examples mainly related to security. Chapter 4 introduces our notions of gray-box composition and aspect-aware interfaces. We also review how our model fits the requirements derived for the aspect model that have been previously defined based on the application scenarios. Chapter 5 presents some related work and Chapter 6 concludes with future work presentation.

Chapter 2

From the Functional Model to the Security Model

Service-Oriented Computing is based on interactions between distributed services. A service is implemented by an agent, using a technology that can vary following a wide spectrum, from traditional languages like Java to dedicated orchestration languages like BPEL (Business Process Execution Language). However, the technology used is hidden: the agent is a true black box, with only a public interface. To interact, services exchange messages over a network in an asynchronous way, which corresponds to a message-passing model: a message is passed from the emitter agent to the network and then from the network to the receiver agent.

In this chapter, we present a functional model for Service-Oriented Computing, based on a chemical abstract machine, and equip the model with a sound type system. Soundness results from a combination of static checks and dynamic checks, enforced by a control layer wrapping each agent. In order to deal with security issues, we then propose a variation of the functional model. We extend the control layer to allow reference monitors to be defined. We add gray-box capabilities to increase the power of the reference monitors, but limit their scope since we consider a realistic situation where agents and channels can be trustworthy or not.

2.1 The Chemical Abstract Machine

Our functional model belongs to the class of *message-passing models* [17]: agents send and receive messages by using a buffer and without sharing memory or without synchronizing the sending and the receiving of messages in a rendez-vous. Following Lamport and Lynch's criteria [17], we outline its main features.

Communication is assumed to be completely asynchronous: in the absence of failure, messages are eventually delivered but no assumption is made about the delivery duration. If a form of synchrony matters between the sending and the receiving of messages, time constraints should be added to the semantics of the model. The buffer used to communicate models the network. We consider it as a finite multiset of messages, with no bound and no order. Actually, a message is defined as some content on a channel. The channel determines the unique target of the mes-

sage. If the order of messages matters, the content must be enriched with some timestamp and data structures to bufferize messages. The model deals with communication failures and agent failures in a simple way: messages can be lost, that is not delivered, and agents can stop. Adding other kind of failures in the model would involve semantic modifications. Agent executions are also completely asynchronous: agents are truly concurrent, with each its own execution time. Exchanging messages is therefore the only way to coordinate agents. Initially, coordination is only possible between agents that share a channel, for instance between a server providing a channel and its clients, requiring the same channel. Gradually, agents discover new channels since messages can contain channels: channel mobility makes the network topology evolve.

In the following, we precisely describe the formal model, dealing with the communication of messages. We abstract away the content of messages, by using a simple structure, which will be refined in the next section, with a type system. We assume given a set $\mathcal{U}$ of values, a set $\mathcal{K}$ of channels and a function $\mathrm{K} : \mathcal{U} \to 2^{\mathcal{K}}$, mapping each value u to the set $\mathrm{K}(u)$ of channels occurring in u. This assumption suffices to account for channel mobility. The set $\mathcal{K}$ is assumed to be finite, which entails that here is no dynamic creation of new channels. But it is not a limitation, as channel creation can be encoded, by describing a new channel as an existing channel parametrized with a nonce. The two main requirements of our model, asynchronous communication and true concurrency, has led us to resort to a chemical model [7], which is now acknowledged as a standard presentation for such models. The operational semantics of our model is therefore given by a chemical abstract machine. To describe it, we prefer to use a physical terminology, instead of the chemical one, less meaningful here as we will see. We start with the syntactic part, which describes processes built from agents, then particles and the aether containing them, corresponding to molecules and the chemical solution. Semantics rules then describe how the aether evolve: there are structural rules and reduction rules.

2.1.1 Processes and Agents

Processes are described in Table 2.1. A process is described as a set of agents in parallel. An agent has a name a, a state σ and an interface I. The state of agents is kept abstract, in accordance with the black-box principle. Different formalisms, like process algebras, could therefore be used to model agents. An interface declares channels, just by giving their names. Input channels k^{I} corresponds to the channels provided by the agent: input messages are received on these channels. Output channels k^{o} corresponds to the channels that go out of the agent, either as the message channel or as a component of the message content. A process must satisfy two simple rules to be well-formed: (i) Agent Identification – given an agent name a, there is at most one agent with name a, (ii) Channel Univocality – given a channel k, there is at most one occurrence of input channel k^{I}.

2.1.2 Aether

A global process is deployed in the aether, the chemical solution, becoming a heavy static particle. The particle can decompose into smaller ones, still static. During aether's evolution (by reduction), light mobile particles appear: they correspond to output messages emitted by agents,

Process	$p ::= a[\sigma][I]$	Agent a with State σ and Interface I
	$\mid p \parallel p$	Processes in Parallel
Agent	$a \in \mathcal{A}$	Set of Agents
State	$\sigma \in \Sigma$	Set of States
Interface	$I ::= 0$	Empty Interface
	$\mid I \,\&\, c_{io}$	Interface Compound with Channel
Channel	$c_{io} ::= c_{in}$	Input Channel
	$\mid c_{out}$	Output Channel
	$c_{in} ::= k^{\iota}$	Input Channel k
	$c_{out} ::= k^{o}$	Output Channel k
	$k \in \mathcal{K}$	Set of Channels

Table 2.1: Processes and Agents

messages in transit and input messages received by agents. A message is defined as a value on a channel. As different messages can have exactly the same form (same channel, same content), the aether is defined as a multiset $\langle \mu_1, \ldots, \mu_n \rangle$ of particles $\mu_1, \ldots, \mu_n$. Table 2.2 formally sums up this description.

2.1.3 Structural Rules

Structural rules, given in Table 2.3, essentially describe the decomposition of processes in the aether. Formally, they generate a structural congruence for aethers. Applied from left to right, they converge to a normal form, used for reduction. From left to right, they are called fission (or heating) rules, from right to left, fusion (or cooling) rules. In our case, two processes in parallel becomes two particles, and so on, until agents become particles in the aether. Then the interface of each agent is decomposed, to eventually get the agent with its state and a multiset of particles describing the local channels of the agent, either input ones or output ones.

2.1.4 Reduction Rules

Reduction rules describe the irreversible evolution of the aether. They assume that the aether has been transformed by structural rules in a normal form. Their formalization is given in Table 2.4.

[LOC]: Agent a consumes a multiset, possibly empty, of input messages m_{in}, produces a multiset, possibly empty, of output messages m_{out}, and updates its state from σ_1 to σ_2.

Aether	$\Omega ::= \langle\, \overrightarrow{\mu} \,\rangle$	Multiset of Particles μ
Particle	$\mu ::= p$	Process
	$\mid a[\sigma]$	Agent with State
	$\mid a[c_{io}]$	Agent with Channel
	$\mid a[m_{io}]$	Local Message m_{io} at Agent a
	$\mid k(u)$	Message in Transit
Local Message	$m_{io} ::= m_{in}$	Input Message
	$\mid m_{out}$	Output Message
	$m_{in} ::= k^{\iota}(u)$	Input Value u on Channel k
	$m_{out} ::= k^{o}(u)$	Output Value u on Channel k

Table 2.2: Aether

$$p_1 \parallel p_2 \rightleftharpoons p_1 , p_2$$
$$a[\sigma][I \,\&\, c_{io}] \rightleftharpoons a[\sigma][I] , a[c_{io}]$$
$$a[\sigma][\mathsf{0}] \rightleftharpoons a[\sigma]$$

Table 2.3: Aether – Structural Rules

[LOC]	$a[\sigma_1] , \overrightarrow{a[m_{in}]}$	$\rightarrow a[\sigma_2] , \overrightarrow{a[m_{out}]}$	
[OUT]	$a[k^{o}(u)] , a[k^{o}] , \overrightarrow{a[c_{out}]}$	$\rightarrow k(u) , a[k^{o}] , \overrightarrow{a[c_{out}]}$	$(*)$
[IN]	$k(u) , a[k^{\iota}]$	$\rightarrow a[k^{\iota}(u)] , a[k^{\iota}] , \overrightarrow{a[c_{out}]}$	$(*)$

Table 2.4: Aether – Reduction Rules

[OUT]: Agent a sends message $k^{o}(u)$ over the network. A local condition must be met: all the channels occurring in the message ($\{k\} \cup \mathrm{K}(u)$) must be output channels declared by the agent.

[IN]: Agent a receives message $k(u)$ from the network. A local condition must be met: the message channel k must be declared as an input channel by the agent. Moreover, the agent upgrades its declaration of output channels by adding all the channels discovered in the content u of the message ($\mathrm{K}(u)$).

(In Table 2.4, condition $(*)$ means: $\overrightarrow{c_{out}} = \mathrm{K}(u)$.)
Actually, these three rules correspond to axiom schemata. More precisely, the rules [IN] and [OUT] are true schemata, whereas the rule [LOC] is a generic form for specifying schemata: with each agent comes a set of schemata having the form [LOC] and defining the local behavior of the agent. The axioms are therefore parametrized by the definition of agents. Besides axioms, there

Types	$t ::= \mathtt{B}$	Base Types
	$\mid t \times t$	Cartesian Product
	$\mid t + t$	Sum
	$\mid T \mid \mu T.t$	Type Variable or Recursive Definition
	$\mid < t >$	Channel Type
Value	$u ::= b$	Base Value
	$\mid (u,u)$	Ordered Pair
	$\mid \mathtt{l}\, u \mid \mathtt{r}\, u$	Left or Right Injection
	$\mid k$	Channel
Channel	$k \in \mathcal{K}$	

Table 2.5: Data Types and Values

are two inference rules, not recalled here since common to all chemical abstract machines [7], the *Reaction Law* allowing schemata to be instantiated in an aether, and the *Chemical Law* allowing local reactions to be fired in an aether.

We could now state a first soundness property. Assume a well-formed process, satisfying the following property, expressing interface consistency: given an output channel k^o declared by an agent, there is a corresponding input channel k^ι declared by some agent. Then, during its execution, there is no dangling message in the aether, that is no message $k(u)$ without an agent declaring k^ι as an input channel. We do not formally prove this property here as we refine it in the next section, with type soundness. It follows from the conditions about well-formedness, the static check about interface consistency and the two dynamic checks in Rules [IN] and [OUT], as well as the channel discovery in Rule [IN].

2.2 A Sound Type System

We now extend our model by adding a type system. Under some conditions ensured by static and dynamic checks, the type system is sound, as we prove it: "well-types services do not go wrong", we could say, paraphrasing Milner's famous slogan concerning well-typed programs.

2.2.1 Types and Values

In our extended model, agents receive and send messages via typed channels: each channel requires a specific type of data. The types and the corresponding values are described in Table 2.5. For the informational part of data, there are standard constructors, allowing structured data to be composed. For the communication part, there is a specific constructor for channels: channel mobility thus provides an encoding for request-response protocols or discovery protocols. Types

are also recursive, for two reasons. First recursivity increases expressivity, leading to simple encodings of common types like lists. Second channel mobility requires recursivity, as shown for instance by the type system of the polyadic π-calculus [20]. Indeed, a channel can convey a channel, a continuation channel, which in turn can convey a channel, and so on.

More precisely, a type can be a base type, like an integer type or a string type, the Cartesian product or the sum (or disjoint union) of types. It can also be a recursive definition, representing the least fixed point of the associated type operator, when it is non-void. Indeed, we consider types modulo the equivalence induced by the expansion of recursive definitions: $\mu T.t = t\,[\mu T.t/T]$. A type can also be a channel type, allowing channels to be sent over the network. For instance, the channel type $<\texttt{int}>$ is a type for channels conveying integers whereas the channel type $<\mu T.\texttt{int} \times <T>>$ is a type for channels that send an integer and a channel of the same type.

Types are interpreted as sets of values. To each base type B, we assume the existence of a set, also denoted by B, whose members b are the values with type B. All these base sets are assumed to be pairwise disjoint. Given values u_1 and u_2 with types t_1 and t_2 respectively, the ordered pair (u_1,u_2) belongs to the Cartesian product $t_1 \times t_2$ whereas the left injection $\texttt{l}\,u_1$ and the right injection $\texttt{r}\,u_2$ belong to the sum t_1+t_2. Due to the equivalence induced by the expansion of recursive definitions, there is no specific constructor for values of recursive types. Finally, each channel comes with a type, given by some typing environment: channels convey monomorphic values.

2.2.2 Type System

Formally, the interpretation of the closed types is defined by a standard type system, as given in Table 2.6. Judgment $\Gamma \vdash u : t$ means that value u has type t in environment Γ, where t is a closed

$$\frac{(b \in \texttt{B})}{\Gamma \vdash b : \texttt{B}} \qquad \frac{\Gamma \vdash u_1 : t_1 \quad \Gamma \vdash u_2 : t_2}{\Gamma \vdash (u_1,u_2) : t_1 \times t_2}$$

$$\frac{\Gamma \vdash u_1 : t_1}{\Gamma \vdash \texttt{l}\,u_1 : t_1+t_2} \qquad \frac{\Gamma \vdash u_2 : t_2}{\Gamma \vdash \texttt{r}\,u_2 : t_1+t_2} \qquad \frac{(\Gamma(k) = t)}{\Gamma \vdash k : <t>}$$

Table 2.6: Type System

type and Γ is a partial mapping assigning a closed type to each channel occurring in u. Precisely, the interpretation of a closed type t is the set of values u for which the typing judgment $\Gamma \vdash u : t$ is valid, where the typing environment Γ declares the type associated to all existing channels. Note that the type system assumes a specific form for each closed type, because of the rule for channels: each channel type that is not nested in the definition of another channel type is also closed, which is always possible to get by expansion.

In the following, the type system is mainly used to check that some given value has some given type. We deduce from the inference system in Table 2.6 that this type checking is clearly computable in polynomial time with respect to the size of the type. A by-product of this checking

is the inference of the types of the channels occurring in the value. This inference is deterministic, as shown by the following proposition.

Proposition 1: Deterministic Type Inference for Channels Assume value u has type t in both environments Γ_1 and Γ_2. Then for any channel $k \in \mathcal{K}(u), \Gamma_1(k) = \Gamma_2(k)$. □

The proof is straightforward, by induction over the proof of the typing judgment.

This type inference plays a fundamental role. Indeed, when an agent receives a message, it can discover new channels in its content. The type inference then allows the agent to discover not only the name of the channel but also its type.

In order to accommodate the type system, we update the model, by modifying the definition of agent interfaces: local channels are declared not only with their names but also with their types.

$c_{in} ::= k^{\iota}(t)$	Input Channel k with Type t
$c_{out} ::= k^{o}(t)$	Output Channel k with Type t

2.2.3 Type Soundness

To elaborate the type soundness property, we need to introduce a notion of error, involving both typing and communication. An error happens when a message is *dangling* in the aether: no agent can receive this message, because either there is no corresponding input channel, or there is one, but waiting a value with a different type.

Definition 2: Dangling Message A message $k(u)$ in transit in aether Ω is *receivable* if there are some agent a, some type t and some typing environment Γ such that agent a declares input channel k^{ι} with type t and value u has type t in environment Γ:

$$\exists a, t, \Gamma \,.\, a[k^{\iota}(t)] \in \Omega \text{ and } \Gamma \vdash u : t.$$

A message is *dangling* if it is not receivable. □

In order to avoid dangling messages, we add checks, static and dynamic. First, a static check evaluates interface consistency for each process: each output channel declared must be associated to a corresponding input channel, with the same type. Second, two dynamic checks are added to the communication rules [OUT] and [IN]. They ensure that the value emitted is well-typed and that the value received is well-typed. Moreover, at reception, they allow the type of the discovered channels to be correctly inferred. Type soundness follows from these checks, as we formally prove.

Theorem 3: Type Soundness Consider a well-formed process. Assume interface consistency: given an output channel $k^{o}(t)$ declared in an interface with type t, there is a corresponding input channel $k^{\iota}(t)$ declared in an interface with the same type t. Assume the rules [OUT] and [IN] are defined as follows:

$$\text{[OUT]}\ a[k^{o}(u)], a[k^{o}(t)], \overrightarrow{a[c_{out}]} \rightarrow k(u), a[k^{o}(t)], \overrightarrow{a[c_{out}]} \quad (*)$$

$$\text{[IN]}\quad k(u), a[k^{\iota}(t)] \rightarrow a[k^{\iota}(u)], a[k^{\iota}(t)], \overrightarrow{a[c_{out}]} \ (*)$$

where condition $(*)$ is[1]

$$\mathrm{dom}(\overrightarrow{c_{out}}) = \mathrm{K}(u) \text{ and } \overrightarrow{c_{out}} \vdash u : t.$$

Then, during the execution of the process, there is no dangling message in the aether. □

Proof: Consider a well-formed process p and the associated aether $\Omega = \langle\, p \,\rangle$. We prove by induction over any sequence of reductions of aether Ω that (A) aether Ω is structurally equivalent to a well-formed process satisfying the interface consistency property and (B) for each message $k(u)$ in transit in aether Ω, there exists a typing environment Γ and a type t such that (i) the domain of typing environment Γ is the set of channels occurring in u: $\mathrm{dom}(\Gamma) = \mathrm{K}(u)$, (ii) value u has type t in environment Γ, (iii) any typed channel occurring in Γ is declared as a typed input channel in an interface: $\forall k', t' . \Gamma(k') = t' \Rightarrow \exists a . a[k'^{\mathrm{I}}(t')]$. We can then conclude since the second condition (B) implies that any message in transit is receivable.
The proof is straightforward for cases [LOC] and [OUT], and for case [IN] by using Proposition 2.2.2. □

To conclude, the static check ensuring interface consistency and the dynamic checks regulating communication rules allow dangling messages to be avoided. If a local agent produces a message with a wrong type, then the message is blocked, without being communicated. The error is detected where it first occurs. When an agent receives a message, it can check its content against the channel type, inferring the type of the channels contained in the message. Thanks to interface consistency and to the emission check, this inference is necessarily correct.

2.3 Variations for Security

Type soundness makes visible a control layer, used to dynamically check messages. To define the security model, we generalize this layer, by allowing more control. We also limit the scope of control, by considering a realistic situation where agents are controllable or not, and where channels are secure or not.

2.3.1 The Control Layer

In order to enforce security policies, we propose to add a control layer, at the communication level. The control mechanism is based on *distributed reference monitors*. Each reference monitor encapsulates an agent and mediates all the communications with other agents: the invocation of services, which was completely free since services are public, can now become controlled. In order to be secure, a reference monitor must satisfy three criterion [3]:

- Non-Bypassability Criterion: a reference monitor is always invoked during each communication,
- Tamperproofness Criterion: a reference cannot be modified by unauthorized agents,

[1] Any sequence $\overrightarrow{c_{out}}$ of typed output channels occurring in a well-formed process satisfying the interface consistency property defines a functional relation from channels to types, and therefore can be considered as a typing environment, which is assumed in the following. Its domain is denoted $\mathrm{dom}(\overrightarrow{c_{out}})$.

- Verifiability Criterion: a reference monitor can be proved correct, with a sufficient level of trust.

With such a control mechanism, only communication messages between agents at the service level are controlled. Is it sufficient to enforce a given security policy? First, to ensure the non-bypassability criterion, an extra assumption is required: there is no covert channel between agents. Second, since only messages between agents are monitored either at emission or at reception, the security targets involved in the security policy must be defined at the level of services. Thus, when an agent manipulates a sensitive resource, the resource must be visible in the interface declared by the agent. Since in the general case the agent does not need to provide to other agents services to manipulate the resource, we must split services into two parts: *external services* that are explicitly invoked by other agents, and *internal services* that are implicitly invoked by the agent itself. Whereas an explicit invocation corresponds to a message generated by an agent, the implicit invocation of an internal service corresponds to the internal manipulation of the resource, which is hidden. In order to intercept the manipulation, we propose to use aspects in order to generate the messages controlled and the subsequent actions of the reference monitor. Thus, agents, which were black boxes, now become gray boxes, exposing some of their internal structure.

For instance, suppose that an agent manages some resource. First, assume that the agent sends a request to the resource. The request is intercepted by the aspect, firing the control of the reference monitor. It is then transmitted to the resource, to be processed after the control. Second, when the resource sends its reply to the agent, the reply is intercepted by the aspect, again firing the control of the reference monitor. It is finally transmitted to the agent, after the control.

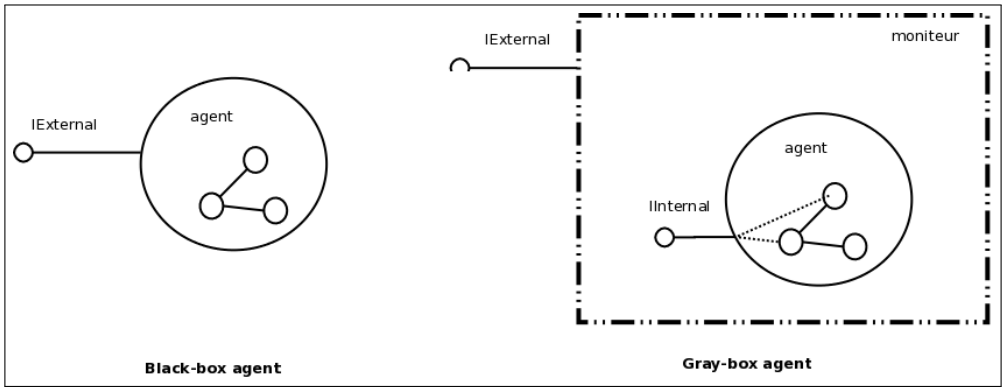


Figure 2.1: Black-box and Gray-box Views

2.3.2 Scope for Control Layer

In the functional model, all agents were assumed to be controlled. For instance, type soundness assumes dynamic checks for each agent. This assumption is unrealistic: some agents can be out

of control and turn out to be malicious. In addition, some channels used to communicate can be insecure, allowing potential attacks during communication: replay attacks, confidentiality and integrity violations. For these reasons, we refine the security model by limiting the scope of the control layer in both directions, agents and channels.

The idea of security domain is important to define the trustworthiness of agents in a collaboration. Thus we split the agents into two disjoint domains: *uncontrolled* ones and *controlled* ones. Uncontrolled agents typically represent third party that are not trustworthy: they can fake messages, change identity, try to decrypt information, and so on. On the contrary, controlled agents behave in a constrained way, in order to enforce security properties, since they are wrapped by reference monitors that apply basic control mechanisms. Thus, distributed reference monitors are at the core of our security model.

Since only the controlled agents can be monitored, four communication scenarios are possible, as shown in Figure 2.2:

1. Between two controlled agents,
2. From a controlled agent to an uncontrolled one,
3. From an uncontrolled agent to a controlled agent,
4. Between two uncontrolled agents.

The fourth scenario (between two uncontrolled agents) is not pertinent from a control point of view, since several uncontrolled agents can be considered as a single uncontrolled agent. As also described in Figure 2.2, a control mechanism is generally enforced by adapting four communications rules, respectively denoted by [OUT$_{\mathcal{C}\to\mathcal{C}}$], [OUT$_{\mathcal{C}\to\mathcal{U}}$], [IN$_{\mathcal{C}\to\mathcal{C}}$] and [IN$_{\mathcal{U}\to\mathcal{C}}$], each rule involving a controlled agent, either as an emitter, as a receiver, or as both. If the controls applied at emission by Rules [OUT$_{\mathcal{C}\to\mathcal{C}}$] and [OUT$_{\mathcal{C}\to\mathcal{U}}$] are different, the reference monitor must be able to discriminate the output messages according to their receiver: this is indeed the case, since we assume that the emitter knows the receiver. If the controls applied by Rules [IN$_{\mathcal{C}\to\mathcal{C}}$] and [IN$_{\mathcal{U}\to\mathcal{C}}$] are different, the reference monitor at reception must also be able to discriminate the input messages according to their emitter: authentication is then required to warrant discrimination.

Besides agents, which may be hostile, channels can also be insecure. For instance, a channel can go through the Internet and use an hostile router. In the following, an insecure channel is modeled as a path from the emitter to the receiver via a router, an extra agent acting as the attacker. We only consider insecure channels between controlled agents since a channel from or to an uncontrolled agent is insecure by definition. Two variants of the communication rules are used for insecure channels between controlled agents: they are denoted [OUT$_{\mathcal{C}\to\mathcal{U}_{\text{router}}}$], and [IN$_{\mathcal{U}_{\text{router}}\to\mathcal{C}}$].

Finally, to sum up, we use the following definitions and associated notations. We qualify the channels as either *controlled*, *uncontrolled* or *insecure*. A channel that is not insecure is said:

- controlled if its target (an agent) is controlled,

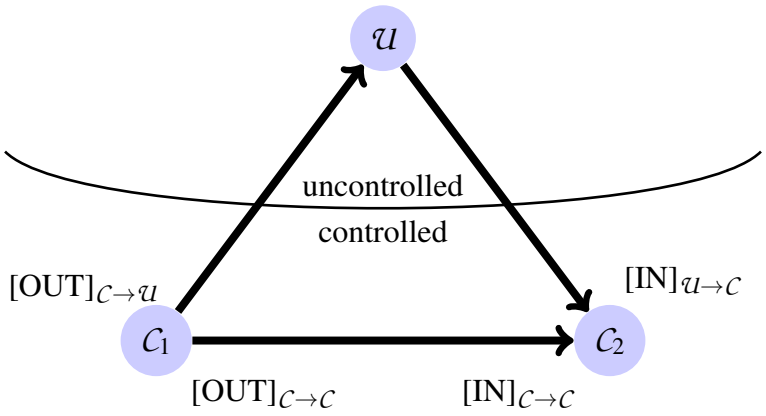


Figure 2.2: Controlled and Uncontrolled Agents - Scenarios and Rules

- uncontrolled if its target (an agent) is uncontrolled.

The agent kinds are described as follows.

$\mathcal{A}^C$: controlled agents.
$\mathcal{K}^U$: uncontrolled agents.

The channel kinds are described as follows.

$\mathcal{K}^C$: controlled channel.
$\mathcal{K}^U$: uncontrolled channel.
$\mathcal{K}^I$: insecure channel.

We also assume that the kind of a channel is statically declared and can be determined from the channel name.

Chapter 3

Control Mechanisms

In the preceding chapter, we have described the functional model and a simple but necessary variation to get a security model with a stong control layer based on reference monitors. We now describe an aspect language for defining reference monitors.

Reference monitors wrap agents to enforce security properties. By their local controls, they enforce a global security property, often specified in an abstract way with a choreography. In this deliverable we are only interested in means for enforcing local properties, without considering the relationship between local controls and global properties. We let the projection theory to a future work.

The aspect language used to implement reference monitors allows communications rules to be transformed: the communication rules [IN] and [OUT] are modified, in order to filter or to rewrite messages or to produce new messages. This simple view seems sufficiently powerful to address all the issues regarding security we are interested in, as shown by some examples detailed in Section 3.2.

Because of distribution, to ensure security properties we often need to resort to cryptographic techniques, as described in Table 3.1.

Nonce:	n	number used once
Secret:	`s`	number known in some domain, unknown elsewhere
MAC:	`H`	function computing a Message Authentication Code
Encrypt/Decrypt:	`enc/dec`	pair of cryptographic functions used to code/decode messages

Table 3.1: Security Related Hypotheses

3.1 The Aspect Language

3.1.1 IN and OUT New Rules

We will define an aspect language to statically weave security aspects on the rules we have defined in the functional model. Thus we have as base program the semantics rules, as pointcut predicates related to a rule and an advice. The precise definition of the advice language is left open, since we need further investigations to choose a definitive syntax. In the sequel we present the principles behind pointcuts and advices. The next section will illustrate the result of applying some aspects to the semantics rules.

In our model a communication rule is identified by an agent which is the target of this rule, a channel allowing to carry the data and a qualifier (IN or OUT) for the direction of the message. Informally, the weaving mechanism, consists in selecting a rule, filtering its context with the pointcut and applying the advice.

$$\overline{Rule\ r ::= < a, k, (IN|OUT) >}$$

Table 3.2: Rule Identification

The new rules only apply to controlled agent by definition, this information is implicit in the rules. That means that for an [OUT] the emitter agent is a controlled agent and for an [IN] the receiver agent is controlled. Indeed we do understand that these new rules apply on an agent with its reference monitor. The general form of these rules is an IN or an OUT rule, with additional conditions (the pointcut) and additional actions and/or modifications of the initial actions (the advice).

Uncontrolled agents can forge any kind of messages. We may also have several rules with the same left hand side but with different message formats, for instance for authentication. Hence we assume pattern matching on the messages to select the right rule to apply. Very often an aspect has an effect on the IN rule and a reverse effect on the OUT rule. This is for instance the case for most of the examples in the next section. But the logging aspect is one which applies on OUT rules only.

3.1.2 Pointcuts formal language

The basic principle for a pointcut is to intercept incoming and outgoing messages. The monitoring of messages needs, in several cases, a notion of state, for instance capturing the message history. Thus our pointcut language is simply a predicate language on states. We can check channels, values, identities, secret information, hash keys, etc. This pointcut language can be used to catch outgoing as well as incoming message of a process. Some concrete examples of predicates are: testing the kind of a channel, testing a value, verifying a hash key, checking the type of a message, etc.

Predicate	$p,q ::= True \mid False$	Base predicates
	$\mid P(\sigma)$	State predicates
	$\mid p\ AND\ q \mid p\ OR\ q \mid NOT\ p$	Boolean operations

Table 3.3: Pointcut Language

3.1.3 Actions language

The actions or advices of our aspects should allow to remove a message, to produce new messages to other partners or to rewrite some parts of the messages. For instance, a logging aspect is a simple copy of messages, while integrity of some data needs to modify the message contents. It is possible to check the request and the correlation data in a message to ensure that the message is correctly routed to the right process of a conversation and in the right ordering. Yet the state of the monitor or the past history is still an important feature, for instance in case of protocol based behavior. These actions could take place at both sides of the process. But we have a system which is able to discover channels, to pass channels to other, and it could be useful to invalidate or to prohibit the use of some channels. To summarize we need constructions to:

- generate, filter or modify messages,
- change the monitor state and
- manage its set of known channels.

The advice language can be seen as a transformation language for a multiset of particles (the original right part of the rule). It should add new particles, forget some particles, or modify the original particles. Formally it can be defined as a function with domain and range a multiset of particles.

3.2 Some Examples

This section describes several different examples illustrating the flexibility of our model.

3.2.1 Logging

This monitoring is really simple and its description in our formal model is simple too. The communication rules [OUT] and [IN] have to be modified for all controlled agents in such a way that each output or input message is duplicated and sent to some logging server. The rule [OUT] is depicted in Table 3.4, where `log` is the channel used to log. We assume that the channel is known by all the reference monitors but is not known by the agents. Furthermore we assume that the channel is secure, otherwise it must be secured using a nonce and a signature as explained in the next section.

[OUT]	$a[k^o(u)], a[k^o], \overrightarrow{a[c_{out}]}$	$\rightarrow$ $\overrightarrow{c_{out}} = \mathrm{K}(u)$	$k(u), a[k^o], \overrightarrow{a[c_{out}]}, \mathtt{log}(k,u)$

Table 3.4: Logging

3.2.2 Authentication

Assume that the security policy requires each message to be signed by the sender. Thus each agent a should add to each message that it sends its identity id_a. But some controls are needed to ascertain the identity of the sender of a message, since there are clear authentication problems. At emission, a misbehaving agent, either dishonest or clumsy, can fake its identity. A reference monitor controlling the agent can easily correct this misbehavior, by emitting over the network only the messages with a right signature. At reception, the reference monitor controlling the receiver agent must discriminate between messages sent by another reference monitor, with a right signature, and messages sent by an uncontrolled agent, with a possible faked signature. A solution is to add to the signature a secret, only known by reference monitors. The authentication controls are depicted in Figure 3.1. Table 3.5 details the communication rules with controls en-

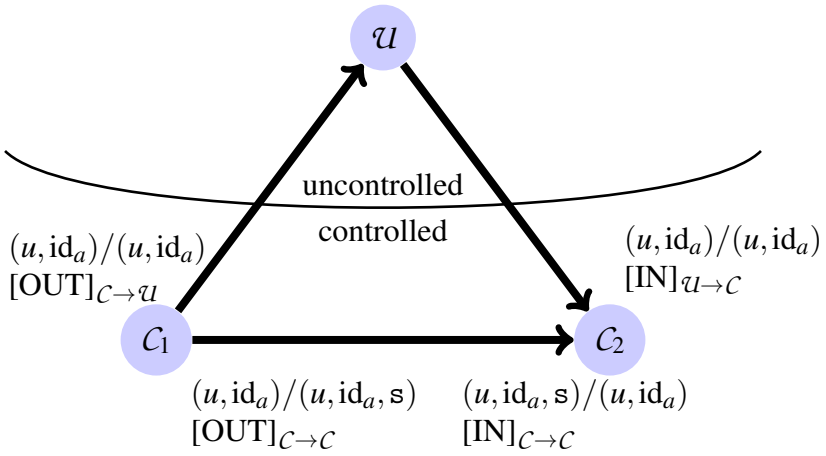


Figure 3.1: Authentication – Secure Channels

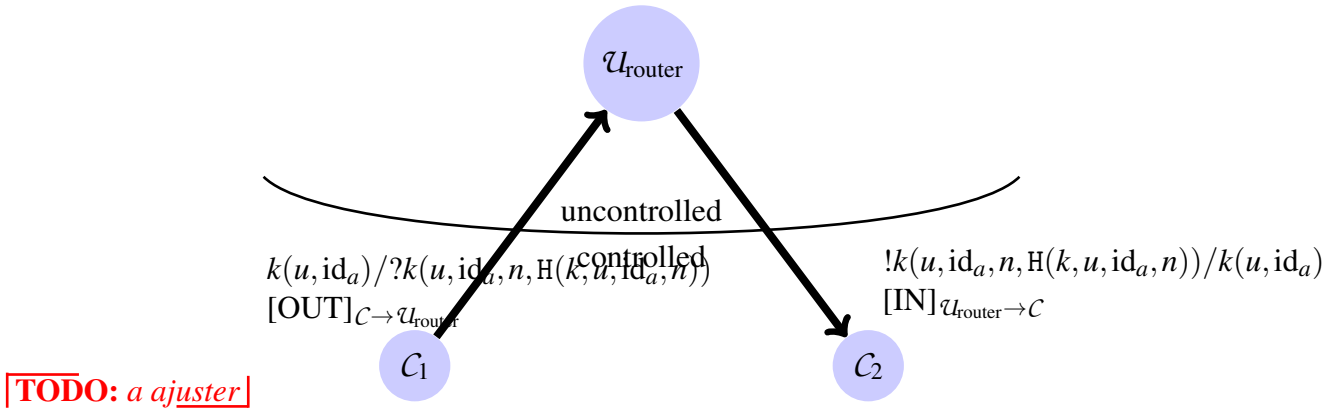
forcing authentication. We assume that all the reference monitors share a function that assigns to each channel its status, controlled or uncontrolled. Thus, the channels that are discovered can be qualified as controlled or uncontrolled (belonging to $\mathcal{K}^C$ or to $\mathcal{K}^U$ respectively), which is requisite for rules [OUT]. The two rules [OUT] allow messages to be sent over the network, after an identity check. When the target is a controlled agent, the secret $\mathtt{s}$ is added to the message. The two rules [IN] allow messages to be received from the network, after an authentication check. If the message has been enriched with the secret, it necessarily comes from another reference mon-

itor, and therefore contains the right identity. If the message has not been enriched, it necessarily comes from an uncontrolled agent, then identified in a generic way by the identity $\perp$.

$[\text{OUT}]_{\mathcal{C}\to\mathcal{C}}$	$a[k^o(u,\text{id}_a)], a[k^o], \overrightarrow{a[c_{out}]}$	$\rightarrow$ $\overrightarrow{c_{out}} = \text{K}(u)$ $k \in \mathcal{K}^C$	$k(u,\text{id}_a,\mathtt{s}), a[k^o], \overrightarrow{a[c_{out}]}$
$[\text{OUT}]_{\mathcal{C}\to\mathcal{U}}$	$a[k^o(u,\text{id}_a)], a[k^o], \overrightarrow{a[c_{out}]}$	$\rightarrow$ $\overrightarrow{c_{out}} = \text{K}(u)$ $k \in \mathcal{K}^U$	$k(u,\text{id}_a), a[k^o], \overrightarrow{a[c_{out}]}$
$[\text{IN}]_{\mathcal{C}\to\mathcal{C}}$	$k(u,i,\mathtt{s}), a[k^\iota]$	$\rightarrow$ $\overrightarrow{c_{out}} = \text{K}(u)$	$a[k^\iota(u,i)], a[k^\iota], \overrightarrow{a[c_{out}]}$
$[\text{IN}]_{\mathcal{U}\to\mathcal{C}}$	$k(u,i), a[k^\iota]$	$\rightarrow$ $\overrightarrow{c_{out}} = \text{K}(u)$	$a[k^\iota(u,\perp)], a[k^\iota], \overrightarrow{a[c_{out}]}$

Table 3.5: Authentication Rules – Secure Channels

The solution presented in Figure 3.1 and in Table 3.5 is valid when the channels are secure. Assume now that a channel is insecure. Then the secret $\mathtt{s}$ can be disclosed and used to compromise authentification. For an insecure channel, we need to use a *message authentication code* (MAC), which amounts to make the secret univocally dependent on the message. Thus, if the MAC is disclosed, it cannot be used except in the same message. Figure 3.2 presents the controls enforced with a non secure channel while Table 3.6 details the communication rules. An insecure channel k is modeled as a pair $?k$ and $!k$ of channels and an intermediate agent, a router potentially hostile. Channel $?k$ is used between the sender and the router and channel $!k$ between the router and the receiver. Function $\mathtt{H}$ computes the MAC from the channel k, the value u, the identity id_a of the agent and a nonce n (the value of a counter associated to each insecure output channel). To ensure security, we assume that the following problem is hard to compute: given a tuple (k,u,i,n), find h such that $h = \mathtt{H}(k,u,i,n)$. In other words, MACs are unforgeable. At emission, the reference monitor adds the MAC to the message. The potentially hostile router receives the message, with its MAC. Since MACS are unforgeable, it cannot use the MAC in another message. At reception, the reference monitor checks that the MAC corresponds to the message, which ensures authentication. Nonces are useful here to avoid a replay attack, where the router would send twice the same message. The nonce manager belongs to the state of the reference monitors. This is a first example for stateful reference monitors.



TODO: *a ajuster*

Figure 3.2: Authentication – Insecure Channels

$[\text{OUT}]_{\mathcal{C}\to\mathcal{U}_{\text{router}}}$	$a[k^o(u,\text{id}_a)]\,,$ $a[k^o]\,,\overrightarrow{a[c_{out}]}\,,$ $a[k^o(n)]$	$\to$ $\overrightarrow{c_{out}} = \text{K}(u)$ $k \in \mathcal{K}^I$	$?k(u,\text{id}_a,n,\text{H}(k,u,\text{id}_a,n))\,,$ $a[k^o]\,,\overrightarrow{a[c_{out}]}\,,$ $a[k^o(n+1)]$
$[\text{IN}]_{\mathcal{U}_{\text{router}}\to\mathcal{C}}$	$!k(u,i,n,h)\,,$ $a[k^\iota]\,,$ $a[k^\iota(N)]$	$\to$ $\overrightarrow{c_{out}} = \text{K}(u)$ $h = \text{H}(k,u,i,n)$ $n \notin N$	$a[k^\iota(u,i)]\,,$ $a[k^\iota]\,,\overrightarrow{a[c_{out}]}\,,$ $a[k^\iota(N\cup\{n\})]$

Table 3.6: Authentication Rules – Insecure Channels

3.2.3 Type-Checking

The type system given in Section 2.2 is sound, provided that all the agents are controlled. Indeed, assume an uncontrolled agent, or an insecure channel, which is not different since its hostile actions are modeled with an uncontrolled router. Given a controlled channel k with type t, the agent can put channel k inside a value u, instead of a channel with some other type t' different from t, so that the value, initially well-typed, becomes ill-typed. Assume the uncontrolled agent sends value u to a controlled agent a that does not know channel k. When agent a checks the typing of value u, it also infers the type of the channels that it discovers. Therefore it infers that channel k has type t', which is a wrong inference since k has type t. If later agent a sends a

message over channel k with type t', the message becomes dangling: the type system is unsound.

The solution is to prohibit the discovery of controlled channels in the messages coming from an uncontrolled agent. Since identification is required, authentication is still needed. Tables 3.7 and 3.8 detail the new rules for typing. The rules [OUT] and [IN] add authentication to the

$[\text{OUT}]_{\mathcal{C}\to\mathcal{C}}$	$a[k^o(u)]\,, a[k^o(t)]\,, \overrightarrow{a[c_{out}]}$	$\to$ $\text{dom}(\overrightarrow{c_{out}}) = \text{K}(u)$ $\overrightarrow{c_{out}} \vdash u : t$ $k \in \mathcal{K}^C$	$k(u, \mathtt{s})\,, a[k^o(t)]\,, \overrightarrow{a[c_{out}]}$
$[\text{OUT}]_{\mathcal{C}\to\mathcal{U}}$	$a[k^o(u)]\,, a[k^o(t)]\,, \overrightarrow{a[c_{out}]}$	$\to$ $\text{dom}(\overrightarrow{c_{out}}) = \text{K}(u)$ $\overrightarrow{c_{out}} \vdash u : t$ $k \in \mathcal{K}^U$	$k(u)\,, a[k^o(t)]\,, \overrightarrow{a[c_{out}]}$
$[\text{OUT}]_{\mathcal{C}\to\mathcal{U}_{\text{router}}}$	$a[k^o(u)]\,,$ $a[k^o(t)]\,, \overrightarrow{a[c_{out}]}\,,$ $a[k^o(n)]$	$\to$ $\text{dom}(\overrightarrow{c_{out}}) = \text{K}(u)$ $\overrightarrow{c_{out}} \vdash u : t$ $k \in \mathcal{K}^I$	$?k(u, \text{id}_a, n, \mathtt{H}(k, u, \text{id}_a, n))\,,$ $a[k^o(t)]\,, \overrightarrow{a[c_{out}]}\,,$ $a[k^o(n+1)]$

Table 3.7: Type-Checking – Rules [OUT] with Authentication

original type checks, following the techniques presented in Section 3.2.2. Discovery is also restricted. In Rule $[\text{IN}_{\mathcal{U}_{\text{router}}\to\mathcal{C}}]$, the channel discovered must be uncontrolled, which avoids a wrong inference for controlled or insecure channels. Again, the reference monitors are stateful, because of nonce management.

3.2.4 Integrity and Confidentiality

For a controlled channel that is insecure, the security policy can require integrity and confidentiality for messages: integrity means that the router (modeling the potential attacks over the channel) cannot modify the content of the message while confidentiality means that the router cannot access the content of the message.

A traditional solution for integrity is to add to the message a message authentication code (MAC), as seen for authentication in Section 3.2.2. As for confidentiality, the solution is to resort to encryption, with a pair $(\mathtt{enc}, \mathtt{dec})$ of inverse functions to code and decode messages

$[\text{IN}]_{\mathcal{C}\to\mathcal{C}}$	$k(u,\mathtt{s}), a[k^{\iota}(t)]$	$\to$ $\mathrm{dom}(\overrightarrow{c_{out}}) = \mathrm{K}(u)$ $\overrightarrow{c_{out}} \vdash u : t$	$a[k^{\iota}(u)], a[k^{\iota}(t)], \overrightarrow{a[c_{out}]}$
$[\text{IN}]_{\mathcal{U}\to\mathcal{C}}$	$k(u),$ $a[k^{\iota}(t)], \overrightarrow{a[c^1_{out}]}$	$\to$ $\mathrm{dom}(\overrightarrow{c^1_{out}}, \overrightarrow{c^2_{out}}) = \mathrm{K}(u)$ $\mathrm{dom}(\overrightarrow{c^2_{out}}) \subseteq \mathcal{K}^U$ $\overrightarrow{c^1_{out}}, \overrightarrow{c^2_{out}} \vdash u : t$	$a[k^{\iota}(u)],$ $a[k^{\iota}(t)], \overrightarrow{a[c^1_{out}]}, \overrightarrow{a[c^2_{out}]}$
$[\text{IN}]_{\mathcal{U}_{\text{router}}\to\mathcal{C}}$	$!k(u,i,n,h),$ $a[k^{\iota}(t)],$ $a[k^{\iota}(N)]$	$\to$ $\mathrm{dom}(\overrightarrow{c_{out}}) = \mathrm{K}(u)$ $\overrightarrow{c_{out}} \vdash u : t$ $h = \mathtt{H}(k,u,i,n)$ $n \notin N$	$a[k^{\iota}(u)],$ $a[k^{\iota}(t)], \overrightarrow{a[c_{out}]},$ $a[k^{\iota}(N \cup \{n\})]$

Table 3.8: Type-Checking – Rules [IN] with Authentication and with Restricted Channel Discovery

respectively. These functions are known by the reference monitors and must be kept unknown elsewhere. To ensure security, we assume that the following problem is hard to compute: given a value u, find u' such that $u' = \mathtt{dec}(u)$. Table 3.9 details the rules for confidentility.

3.2.5 Protocol Based Behavior

Very often the services provided and used by an agent are invoked following a specific *protocol*. The most simple case is a protocol specified by a finite labelled transition system: a transition is fired at the invocation of a service. An interesting variant makes transitions conditional, with conditions depending on the content of the messages. For instance, a transition is fired if the message conveys the right correlation value. Table 3.10 presents a simple example: a reception is enabled if a transition can be fired. The reference monitor has states ρ, ρ', ..., and enforces a protocol specified by labelled transitions $\rho \xrightarrow{k^-} \rho'$. This approach can be easily applied to security, for instance to enforce an authorization policy.

$[\text{OUT}]_{\mathcal{C}\to\mathcal{U}_{\text{router}}}$	$a[k^o(u)], a[k^o], \overrightarrow{a[c_{out}]}$	$\to$ $\overrightarrow{c_{out}} = \text{K}(u)$ $k \in \mathcal{K}^I$	$?k(\texttt{enc}(u)), a[k^o], \overrightarrow{a[c_{out}]}$
$[\text{IN}]_{\mathcal{U}_{\text{router}}\to\mathcal{C}}$	$!k(u), a[k^\iota]$	$\to$ $\overrightarrow{c_{out}} = \text{K}(\texttt{dec}(u))$	$a[k^\iota(\texttt{dec}(u))], a[k^\iota], \overrightarrow{a[c_{out}]}$

Table 3.9: Confidentiality Rules

[OUT]	$a[k^o(u)], a[k^o], \overrightarrow{a[c_{out}]}$, $a[\rho]$	$\to$ $\overrightarrow{c_{out}} = \text{K}(u)$ $\rho \xrightarrow{k^o} \rho'$	$k(u), a[k^o], \overrightarrow{a[c_{out}]}$, $a[\rho']$
[IN]	$k(u), a[k^\iota]$, $a[\rho]$	$\to$ $\overrightarrow{c_{out}} = \text{K}(u)$ $\rho \xrightarrow{k^\iota} \rho'$	$a[k^\iota(u)], a[k^\iota], \overrightarrow{a[c_{out}]}$ $a[\rho']$

Table 3.10: Protocol Enforcement Rules

Chapter 4

Requirements satisfaction of the aspect model

In the following, we review general requirements derived from an analysis of the security requirements of the application scenarios that have been proposed by the industrial partners.

4.1 General Requirements

In the following the *requirements are set in slanted font*, while the explanation of how our aspect model addresses them is set in standard serif font.

Requirement 1: Horizontal and vertical pointcut definitions

The language must provide constructs to clearly define pointcuts addressing horizontal as well as vertical concerns.

In the loan scenario this need is illustrated in several cases. Consider the protection of a signed contract, for instance. A first policy establishes that a bank clerk needs to sign the ranking of a loan request made by a customer - typically altering the service composition at the collaboration level. In parallel, a second policy would require that all proposed bundle solutions made in the banking domain must be logged locally in a secure device, for ensuring accountability - by recording the user Id, the action executed, and a timestamp. The requirement for secure storage typically influences the vertical composition, modifying the infrastructure by adding secure logging support.

The CESSA framework must allow to express these policies by a composition of vertical and horizontal aspects.

As discussed in Chp. 2, our aspect model consists of different components that smoothly fit together to address this requirement. The aspect model defined in Chp. 3 allows the application of aspects to interacting services. This mechanism is principally designed to apply aspects to horizontal service compositions. Vertical service compositions can be modified using the same

aspect model if the are realized themselves in terms of orchestrated or choreographed service invocations. Vertical compositions that are implemented using calls into some middleware can be manipulated using a traditional aspect language, *e.g.,* AspectJ in the case of Java-based middleware.

As defined in Sec. 4.3, all of these interactions are governed by aspect-aware interfaces that impose protocol-based restrictions. Furthermore, the proposed gray-box mechanism, see Sec. 2.3.1, allows the grouping of services and interactions that may involve different services from horizontal and vertical compositions.

Requirement 2: Avoid the fragile pointcut problem

The fragile pointcut problem is a common limitation of aspect systems that denotes the strong dependence of aspects on the names of base entities, such as services. If the names change in any ways, the dependent aspects breaks. In our context the fragile pointcut problem can be defined as breaking the relationship between a security aspect that is already deployed and its target component, after the security aspect undergoes an evolution.

Our aspect model provides two mechanisms that help avoiding dependencies on low-level information, in particular, names of base entities. First, aspectual properties can frequently be defined in terms of behavioral properties such as protocols. Furthermore, using our means for structured gray-box access, service invocations, for instance, may be referred to by location, or another relationship with other services, instead of by name.

Requirement 3: Control weaving ordering

Several aspects may interact in the sense of being applicable at the same execution point. This raises the problem how the application of these aspects is ordered, whether one aspect may disable another one, etc. The requirement can be illustrated by the classical example of signing a message, then encrypting the result, or, encrypting a message, then signing it. Depending on the ordering, different security policies are put in place.Moreover, it would be interesting to have analysis tools that would inform the user about the different consequences of using a given advice code composition.

In our aspect model, two mechanisms provide the necessary control. First, AAIs can be straightforwardly extended by sequentialization and other relationships between, *e.g.,* service that are modified or invoked from specific aspects. Furthermore, they can also accommodate relations between aspects. Second, the monitor-based aspect model will be extended by means for aspect composition as need arises in the context of the application domains.

Requirement 4: White-, gray- and black-box service modification and composition

The aspect language is required to enable aspects to modify service implementations. Several use cases underlying the applications of the CESSA project require either to gather data in order to

produce an access control decision from internals of the service/process definitions, or to change its crosscutting functionalities of its internal behavior, e.g., *to mitigate code injection attacks.*

Furthermore, since we are targeting an aspect model able to handle several abstraction levels, sometimes we will not have access to the code of a given component. In this case, we will use black-box composition. In the case that we can intervene inside process orchestrations, by modifying the workflow, but not the services being composed, gray-box composition is required. Finally, sometimes complete access to service implementations at all abstraction levels, therefore, i.e., *white-box composition facilities are needed.*

Our model explicitly addresses this requirement through flexible gray-box access to service interfaces and implementations, as well as the inclusion of aspects over service implementation whose effects are mitigated using aspect-aware interfaces.

Requirement 5: Formal semantics

The language shall have a formal semantics. This will enable to perform reasoning about the aspect and service compositions, such as verifying if a security property can be ensured by the deployment of a given aspect combination.

All parts of our model that are used for reasoning about aspect properties have formal semantics, notably the ether-based aspect model for service interactions and the automata-based aspect-aware interfaces).

Requirement 6: Service domains or boundaries

Handling domains in the sense of policy domains from the security policy specification language (CSPL) we are developing is essential for correctly delimiting the reach of aspect deployment. In CSPL domains are structured in trees representing the authority hierarchies.

In our model, the gray-box model and AAIs can easily accommodate domains by use of domain-defining relationship for the definition of boxes and conditions on aspect applicability.

Requirement 7: Support for multiple horizontal composition languages

The aspect language must support pointcut designators well adapted to several existing service composition or orchestration languages. One must be able to both determine aspects to be woven in a service orchestration, e.g. in BPEL, or aspects that will somehow modify how a choreography is described in a concrete WS-CDL document. The horizontal composition may consider weaving into different target languages by adding an abstraction layer, via intermediate languages, that would be later translated to a target language. This would be a very different approach with respect to the classical model of a single base language.

Our aspect language for horizontal composition does not depend neither on the technology use (Restful Web Services or WS* Web Services) nor on the language used to implement the language (BPEL or Java for instance). However, the aspect layer used to intercept internal interactions and to reveal internal services depends on the language used to implement the agent.

Requirement 8: Allow for dynamic or static weaving

The language must not impose an execution model, such that an implementation of it can either weave aspects at load or compile time, while another implementation of it can make use of the application source code. Except for black-box composition, where only the external component interfaces are visible.

All parts of our aspect model can support static and dynamic aspect weaving. Reference monitors, in particular, are defined as dynamic controls. They may however also be implemented using static inlining techniques.

Requirement 9: Syntactical requirements

The language should not rely on the specific syntax for a given target language, but to be able to handle as many languages as possible, in all horizontal and vertical abstraction levels. The complexity of handling multiple languages or SOA platforms must be addressed by the aspect engine or compiler.

The aspect model has been careful defined to abstract from syntactic particularities of the base system or concrete aspect languages.

Requirement 10: Context exposition

The pointcut language must expose the application context at matching joinpoints. The context can be static information about the application or runtime information about the value of parameters, variables, etc*. The context exposition expressivity depends on the level the aspect refers to. For instance, the pointcut language allows message exposition when at the service level. At the process level, the pointcut exposes the process definition and the action history.*

The gray-box model, through its notion of internal and external services enables arbitrary context to be exposed as a part of aspects that modify horizontal and vertical compositions.

4.2 Further features for the extension of the Service Model

The requirements below were collected from the CESSA project Deliverable 1.3, Section 4.2. They describe sophisticated aspect mechanisms interesting for our aspect model. The idea is to have a framework that allows security properties to be expressed and enforced. It relies on an aspect layer whose different possible features are enumerated below.

Requirement 11: Atomic aspect mechanisms

These consist in mechanisms that enable the manipulation of individual service execution, as listed below.

- ***Service calls:*** *redirection, modification of arguments, suppression (etc.) of individual calls (including methods calls that are used to implement services).*

- ***State manipulation:*** *direct modification of security-related state (*e.g.,*, databases) and state in service implementations (*e.g.,*, assignment to state variables).*
- ***Exception throwing and handling:*** *fault handling in service choreographies and orchestrations, exception handling in OO or imperative service implementations etc.*

Solution 1: Expressivity issue (bis)

It is possible to define reference monitors managing atomic transactions, based on a compensation mechanism. The same seems possible for exception handling. □

Requirement 12: History-based aspect mechanisms

History-based approaches allow to directly express many security-related functionality, e.g., *information-flow properties, use of cryptographic protocols, and the monitoring for intrusion detection. Furthermore, history-based pointcuts allow to express modifications directly in terms of the collaboration structure (including choreography and orchestration) of our service model. However, several history-based mechanisms, notably data flow pointcuts and pointcuts defined in terms of non-regular structures, are notoriously difficult to implement efficiently if applied to fine-grained frequent execution events.*

- ***Control flow:*** *AspectJ's* `cflow` *[15], Douence et al.'s* `path` *[13]*
- ***Data flow:*** *Masuhara's et. al.'s* `dflow` *[18]*
- ***Finite-state systems:*** *stateful aspects [14], tracematches [4], etc.*
- ***More expressive (non-regular) languages:*** *Viggers et al.'s tracecuts [25], Ha et al.'s VPA-based aspects [21] etc.*

Solution 2: Stateful reference monitors

As said above, stateful reference monitors allow history-based properties to be enforced. □

Requirement 13: Interfaces mediating the effects of aspects

Approaches for interfaces mediating the application of aspects to base programs provide effective and flexible means for controlling the effects of aspects. Since such control is crucial for the enforcement and analysis of (security) properties, the CESSA aspect model will include a notion of aspect-aware interface, probably consisting of a more primitive notion, similar to applicability conditions, and a declarative mechanism using explicitly typed events.

- ***Controlling aspect application:*** *Douence et al.'s applicability conditions [12], Aldrich's open modules [***?***], Skotinitiois et al.'s aspect interfaces [23]*

- ***Explicit event types****: Leavens et al.'s Ptolemy [?] [, AOSD'11], EScala [AOSD'11]*

Solution 3: Extended declaration of interfaces
See above.

□

Requirement 14: CSPL Compliance

The CSPL language is described in deliverable D2.2. Its main features are: single place for a policy, abstract or detailed policies, agnostic and readable.

Solution 4: Need of Projection Allowing the security policy to be defined in a single place needs some future investigations. As such it means that the policy is global and abstract. The projection theory is the missing piece of our puzzle. It must enable to split a global policy into several local policies which should be implemented on the corresponding agent. The various level of details requirement can be addressed using the notion of gray-box already discussed. Our formal model is agnostic of the implementation language. To improve readability is not a difficult issue, with specific annotations and composition means.

□

4.3 Aspect-aware interfaces

Typically, aspect-oriented systems provide expressive means for invasive modifications at a potentially fine level of granularity. In a service-oriented environment, this may include the replacement of a service calls by another one on the level of interactions governed by choreographies or even on the code level. Arguments of exchanged messages may be altered and variable values may potentially be modified that are deeply hidden within service implementations.

Because of this expressive power, the formal definition, verification and enforcement of properties over such aspects systems is, if not impossible, very intricate. Furthermore, corresponding techniques for the formalization of aspect systems, see the corresponding overview in the CESSA deliverable D1.1 ([11], Sec. 4.3.3), are much too elaborate in order to be applied in real-world settings. A principal means to reconcile the expressivity of aspect languages with tractable property definition and reasoning are so-called aspect-aware interfaces (AAIs) that mediate between aspects and base programs.

Add motivation for an abstract parameterized notion of interface.

4.3.1 Goals

As part of the CESSA project, we propose the use of aspect-aware interfaces in order to pursue the following goals:

G1) Service compositions, notably vertical service compositions, and service implementations should be manipulable using expressive aspect mechanisms that can be restricted in order to preserve or enforce security properties.

G2) AAIs should provide a concise and potentially user-friendly way to define security-relevant properties. They can, for example, make explicit predicates that are defined over the execution history of base programs and trigger actions that enforce access control, confidentiality, etc.

G3) Interfaces should support different kinds of property, for example, properties obtained through restrictions expressed using plain finite-state systems or session types.

G4) While we mainly target aspect systems that mostly use localized control mechanisms over distributed services system, non-local control should be supported where necessary.

G5) AAIs should support tool-based property analysis and enforcement of security properties.

As discussed in Sec. 5.1, several preceding approaches to such interfaces have been proposed in order to strongly restrict the effects that aspects may have on base programs. These approaches therefore satisfy partially the first goal above. However, the existing approaches are not appropriate in our settings because of the following reasons:

- None of the previous approaches has proposed AAIs for service-oriented applications and none has been applied to security properties.
- Previous approaches only support one type of restriction on aspects.
- No tool support for (automatic) property analysis and enforcement exist.
- Most previous approaches use sophisticated formal mechanisms and therefore are very difficult to use by laymen in the use of formal mechanisms.

4.3.2 Aspect-aware interfaces for multi-domain/multi-level services

We propose the following notion of aspect-aware interface (AAI):

- An interface includes declarations of *external and internal services* according to the above gray-box composition model.

  The interface therefore may, for example, include declarations of signatures of services that are used in horizontal compositions but also means for the modifications of a state that is part of the service implementations. Furthermore, the declared entities may correspond to parts of a physically distributed system or logical notions of grouping, such as security domains.

- An interface defines *applicability conditions* that restrict the application of aspects to base programs in terms of the execution state of the service system at the point when the aspect is about to be applied.

  A principal means for the definition of such applicability conditions are protocols over the use of interface entities that are expressed using transition systems.

This notion of interface meets the five goals introduced above: protocols, for example, allow the restriction of the applicability of aspects in terms of traces of horizontal compositions and execution events generated by service implementations (our interfaces thus meet goal G1 above); protocols may be defined and represented in different form and at different levels of abstraction, in particular, using abstractions that enable service users to understand them (G2); applicability conditions may be defined using transition systems of different levels of expressiveness, thus supporting different kinds of property (G3); some kinds of protocol can be enforced using only local control strategies, but others also support non-local ones (G4); transition systems and protocols are formal frameworks that enjoy one of the largest number of supporting automatic and semi-automatic tools for their analysis, verification and enforcement (G5).

interfaces based on finite-state systems and security properties

As motivated before we have deliberately not defined precisely which kind of entities can be referenced in AAIs and how applicability conditions are to be expressed.

As an example of how services, aspects and AAIs can be defined and used together, let us consider an evolution scenario that has the following technical characteristics:

- Properties of interest are expressed in terms of plain finite-state transition systems (equivalent to regular expressions), *e.g.,* in order to exploit existing model checkers for property verification.

- The modifications necessary to enforce these properties in a service-oriented base system can be expressed simply in terms of service calls (*e.g.,* at the choreography level) and method calls (*e.g.,* on the implementation level).

- Aspects that apply the modifications use pointcuts defined in terms of single services or calls, advice is strongly restricted: it only performs security relevant tests (that do not modify the system state), alters the security state by using modifications to reference monitors as discussed later in this document, or stops execution of the system.

- Applicability conditions of aspects in AAIs are defined as non-deterministic regular expressions over service and method calls. They may also refer to domains of the system that group parts of the system and defined in terms of the gray-box model introduced above.

Under the additional assumption that the base system, aspects and applicability conditions are expressed and implemented using Java, the corresponding AAIs and aspect weaving modulo applicability conditions can be performed using the AWED system for distributed AOP in Java [19, 6].

```
// Definition of domains con1, con2, uncon

cond maybeUnverifiedSig(Sig c):
   seq(sendTo(m1, c, _) && target(m2) && con1(m2) && uncon(m1) -> s2 | s3,
       s2: verify(c),
       s3: sendTo(m3, c, _) && con2(m3) -> s1
   )
```

a) AAI condition identifying (un)verified certificates

```
before sendTo(m, d, c) && con2(m) && maybeUnverifiedSig(c) { verify(c) };
```

c) Aspect for certificate verification

Figure 4.1: Certificate verification using AAIs and aspects

In order to illustrate the use of aspects and AAIs in the context for the enforcement of security-relevant properties have a look at the example shown in Fig. 4.1. The AAI shown in a) discriminates messages that are sent from a machine `m2` in the controlled domain `con1` to the machine `m1` in an uncontrolled domain. Subsequently, the certificate `c` associated to the message may be verified (*e.g.,* by contacting a digital notar service) or send to a machine `m3` in the controlled domain `con1`.

The aspect shown in subfigure b) that is fired if the AAI condition is in state `s3` then ensures that certificates are verified - but only doing so if verification has not already been taken place.

Chapter 5

Related Work

TODO: *references to be checked*

5.1 Aspect-aware interfaces

Interfaces that mediate between aspects and the base programs they can be applied to are a fundamental idea in the AOSD community. Similar as we do, almost all of these approaches rely on the definition of syntactic or semantic constraints that limit either the applicability of aspects or the effects aspects may have. Some of the existing approaches strive for constraints that are defined in a formal way and enable reasoning about formal properties involving base programs and aspects.

In the following we present the major approaches to aspect-aware interfaces. However, the existing body of work is quite distinct from ours in one or several important respects:

- Previous approaches typically only provide fixed aspect languages and do not support, as we do, classes of aspect languages, as we do through the parameterization with different kinds of transition system.

- In contrast to our approach that enables reasoning about a range of properties, previous approaches only provide few means for formal reasoning that partially support only very limited properties.

  Furthermore, no tool support exist for the analysis and enforcement of previous notions of aspect-aware interfaces.

- Finally, no previous study of aspect-aware interfaces has taken into account requirements from real-world industrial systems. Finite-state transition systems can be used in our case to leverage aspect-aware interfaces with industrial-strength tools.

In the following, we compare four principal approaches to aspect-aware interfaces (arguably the most well-known ones) — AspectJ aspect-aware interfaces [16], Open Modules [1], XPI [24], and Ptolemy [22, 5] — with respect to these requirements.

Approach	Aspect languages	Properties, reasoning	Tool support	Service appl.
AAAI [16]	1: AspectJ-based	ad-hoc, manual	none	none
Open Modules [1]	1: functional	ad-hoc, manual	none	none
XPIs [24]	1: AspectJ	informal pre/post, no	none	none
Ptolemy [22]	1: OOP, events	logical, manual	JML tools	none

Table 5.1: Comparison of main approaches to AAIs[1]

Table 5.1 shows the result of this comparison in succinct form. All of these approaches do not come close to meeting the requirements of the CESSA project: First, none of them support different aspect languages, instead each of them can be used only with one specific aspect language. Ptolemy, while not possessing an aspect language per se, expresses aspect features using one set of object-oriented and event-based mechanisms.

None of the approaches supports different, well-defined classes of properties that can be reasoned about: AAAI and Open Modules support only a small set of ad hoc properties that do not form a well-defined property class; XPIs allow only the informal definition of pre- and post-conditions; Ptolemy supports turing-complete logical specifications of aspect properties that can be reasoned about only manually using the Java Modeling Language (JML).

Furthermore, none of these previous approaches come equipped with or can interface with tools for the automatic property verification. All approaches but Ptolemy do not enjoy any tool support; Ptolemy can be used with tools for JML, none of which supports automatic reasoning, though.

Finally, no approach has been applied to service-oriented systems.

5.2 Aspects for service-oriented architectures

Aspect-oriented languages and systems that can be used for the manipulation of SOAs based on web services have already been discussed previously in deliverable D1.1 [11].

Among these approaches, only AO4BPEL has been applied to the security of web service compositions, more concretely to services of middleware containers and the manipulation of web services that are orchestrated using BPEL [8, 10]. AO4BPEL enables pointcuts to match sequences of web service invocations in BPEL workflows, and insert or replace service invocations using advice. It provides a turing-complete aspect language that with mechanisms similar to AspectJ. No formal definition of AO4BPEL exists and reasoning over security properties could only be performed by adapting manual proof techniques for AspectJ-like languages; however, no such reasoning method and corresponding tools have been proposed.

[1] Information based on the cited papers and Internet search.

Chapter 6

Conclusion

This deliverable presents our formal model of services and its direct application to some elementary security properties. We sketch an aspect language for our semantics rules which is related to the well-known operational notion of reference monitor. The model enables security domains, in a yet simple basic form with controlled and uncontrolled agents using secure or insecure channels. While this is a basic model, our application examples show that some attentions are needed to ensure simple security properties. This deliverable demonstrates that abstraction and rigor are provided as well as an operational way to enforce security properties. Our current model strictly addresses horizontal composition of services. Vertical composition of services can be achieved using a notion of internal interfaces and gray-box view. The aspect-aware interfaces extends the usual notion of aspect with predicates, state based history, local and non local strategies. The aim is to provide a better control on the properties preserved by the weaving process and tools to check these properties.

Future work aim at exploring several tasks mainly related to the aspect model and security enforcement. The first task is to define more precisely our aspect language and aspect-aware interfaces.

A second task is to add an annotation mechanism for security related aspects. For some "standards" aspects like monitoring, authentication, confidentiality, integrity, an annotation mechanism is possible. From these annotations the real aspects to apply can be automatically generated. Our idea is to annotate types of the channels and the values. From these annotations a transformation modifies the type to add necessary security information. For instance, assuming that *int* must be sent with integrity then the corresponding type is $int \times Nonce \times h(\mathcal{K}, Nonce\ int)$. If we need confidentiality it will be $\texttt{enc}(int)$ at emission and $\texttt{dec}(int)$ at receipt. In this way this annotation mechanism realizes a specific projection of the global property associated with the annotation to the local agents (controlled and uncontrolled).

Another important task is to study the projection theory linking a global property and its local consequences on the level of individual agents.

It is not too difficult to see that our aspect language can address all the basic security requirements. For instance if we want authentication and confidentiality, it is not difficult to write the corresponding aspect. But to simply compose the authentication and the confidentiality aspects could not work. Thus the last task is to study the aspect composition, how to schedule aspect and

when such a composition is safe or make sense.

Bibliography

[1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[2] D. Allam et al. Model and formal architecture specification. Deliverable D1.2, CESSA ANR project, no. 09-SEGI-002-01, January 2011.

[3] J. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-7351, Electronic Systems Division, Hanscom Air Force Base, Hanscom, MA, 1974.

[4] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 589–608. ACM, 2007.

[5] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In Paulo Borba and Shigeru Chiba, editors, *10th International Conference on Aspect-Oriented Software Development, AOSD*, pages 141–152. ACM, 2011.

[6] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06)*, pages 51–62. ACM Press, March 2006.

[7] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1), 1992.

[8] Anis Charfi and Mira Mezini. Using aspects for security engineering of web service compositions. In *ICWS*, pages 59–66. IEEE Computer Society, 2005.

[9] Anis Charfi and Mira Mezini. Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, 10(3):309–344, 2007.

[10] Anis Charfi, Benjamin Schmeling, Andreas Heizenreder, and Mira Mezini. Reliable, secure, and transacted web service compositions with AO4BPEL. In *ECOWS*, pages 23–34. IEEE Computer Society, 2006.

[11] R. Douence et al. Survey and requirements analysis. Deliverable D1.1, CESSA ANR project, no. 09-SEGI-002-01, July 2010.

[12] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Gail C. Murphy and Karl J. Lieberherr, editors, *AOSD*, pages 141–150. ACM, 2004.

[13] Rémi Douence and Luc Teboul. A pointcut language for control-flow. In Gabor Karsai and Eelco Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114. Springer, 2004.

[14] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Mehmet Aksit et al., editor, *Aspect-Oriented Software Development*. Addison-Wesley, 2003.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[16] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.

[17] L. Lamport and N. Lynch. Distributed computing: Models and methods. pages 1157–1199. 1990.

[18] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In Atsushi Ohori, editor, *APLAS*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[19] Ismael Mejia, Luis Daniel Benavides Navarro, and Mario Südholt. The awed system, 2011. Home page: `http://awed.gforge.inria.fr`.

[20] R. Milner. *The polyadic pi-calculus: a tutorial*, pages 203–246. Springer-Verlag, 1993.

[21] Dong Ha Nguyen and Mario Südholt. Property-preserving evolution of components using vpa-based aspects. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 613–629. Springer, 2007.

[22] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179. Springer-Verlag, 2008.

[23] Therapon Skotiniotis, Jeffrey Palm, and Karl J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 477–500. Springer, 2006.

[24] K. Sullivan, W.G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with xpis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):1–42, 2010.

[25] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 159–169. ACM, 2004.