UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MODULAR AND SECURE ACCESS CONTROL WITH ASPECTS

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS
MENCIÓN COMPUTACIÓN

RODOLFO ANDRÉS TOLEDO TOLEDO

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
ERIC BODDEN
JOHAN FABRY
ALEJANDRO HEVIA ANGULO

SANTIAGO DE CHILE
MAYO 2014

RESUMEN DE LA TESIS
PARA OPTAR AL GRADO DE
DOCTOR, MENCIÓN COMPUTACIÓN
POR: RODOLFO ANDRÉS TOLEDO TOLEDO
FECHA: 13/05/2014
PROF. GUÍA: ÉRIC TANTER

MODULAR AND SECURE ACCESS CONTROL WITH ASPECTS

Es inevitable que algunas consideraciones sean transversales a una aplicación de cierto tamaño, lo que resulta en código repartido en ella, mezclado con el código de otras consideraciones. Este problema es particularmente severo en el caso de la seguridad: se torna difícil estar seguro acerca de qué tan segura es la aplicación cuando su implemntación esta repartida y mezclada en la base de código. El razonamiento global acerca de la seguridad se vuelve precario.

En esta tesis, consideramos el caso del control de acceso, un pilar fundamental de toda arquitectura de seguridad. El control de acceso resulta ser una consideración transversal con una implementación no modular basada en la inspección del *stack* en tiempo de ejecución en lenguajes como Java y C#. En esta tesis usamos la orientación a aspectos para definir modularmente el control de acceso. Más precisamente, presentamos el diseño e implmentación del control de acceso, incluyendo sus características avanzadas, de una forma modular.

Finalmente, demostramos que esta implementación modular es segura, incluso en presencia de aspectos no confiables. Una implementación modular como esta soluciona los problemas de mantenimiento y evolución producidos por la naturaleza transversal del control de acceso, pero más importante aun, pavimenta el camino para el razonamiento global acerca del control de acceso.

RESUMEN DE LA TESIS
PARA OPTAR AL GRADO DE
DOCTOR, MENCIÓN COMPUTACIÓN
POR: RODOLFO ANDRÉS TOLEDO TOLEDO
FECHA: 13/05/2014
PROF. GUÍA: ÉRIC TANTER

MODULAR AND SECURE ACCESS CONTROL WITH ASPECTS

It is inevitable that some concerns crosscut a sizable application, resulting in code scattering and tangling. This issue is particularly severe for security-related concerns: it is difficult to be confident about the security of an application when the implementation of its security related concerns is scattered all over the code and tangled with other concerns, making global reasoning about security precarious.

In this thesis work, we consider the case of access control, a cornerstone of every security architecture, which turns out to be a crosscutting concern with a non-modular implementation based on runtime stack inspection in mainstream languages such as Java and C#. We make use of aspect orientation for the modular definition of access control. More precisely, we design and implement access control, including the advanced features associated to it, in a modular way.

We demonstrate that this modular implementation is secure, even in the presence of untrusted aspects. A modular implementation alleviates maintenance and evolution issues produced by the crosscutting nature of access control, and, more importantly, paves the way to global reasoning about access control.

Esta tesis está dedicada a mi madre, simplemente por ser la mejor.

Contents

Publications

The following papers were published as a result of this research:

Journal Papers

- Rodolfo Toledo, Ángel Núñez, Éric Tanter, and Jaques Noyé. Aspectizing Java Access Control. IEEE Transactions on Software Engineering, 38(1):101-117, January/February 2012.
- Rodolfo Toledo and Éric Tanter. Access Control in JavaScript. IEEE Software, 28(5):76-84, September/October 2011.

Conference Papers

- Rodolfo Toledo and É Tanter. Secure and Modular Access Control with Aspects. Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013), pages 157-170, Fukuoka, Japan, March 2013. ACM Press.
- Rodolfo Toledo, Paul Leger, and Éric Tanter. AspectScript: Expressive Aspects for the Web. Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010), pages 13-24, Rennes and Saint Malo, France, March 2010. ACM Press.

Chapter 1

Introduction

Complex software systems become unmanageable if they are not modularized. This is the reason why we should *divide and conquer*: decompose the system into modules until they are simple enough to be manageable, and later on compose them to obtain the whole system. Modules are crucial for raising the understandability, maintainability, reusability, and evolvability of software.

One criterion we can follow to modularize the system is what the *separation of concerns* (SOC) [25, 59] principle states: each one of the modules of a given decomposition should target one specific concern. This fosters both the understandability and extensibility of the system. The system is more understandable because each particular concern is localized in one module. The system is also more extensible because each module can be enhanced or replaced without having to worry about other modules.

Even though following the SOC advice allows us to obtain systems that are easier to extend and understand, the designer of the system implicitly imposes a particular view of the system, which needs to be determined at the beginning of the development process. This view can turn out to be inappropriate later on when trying to define and integrate new modules. This anomaly is called the *tyranny of the dominant decomposition* [77]: a concern that does not fit into the initial view of the system (a *crosscutting* concern), ends up being scattered in several modules, and its corresponding code tangled with the code of other modules.

One way to avoid the scattering and tangling problems in practice is through the use of Aspect-Oriented Programming [50]. In this programming paradigm, a crosscutting concern is expressed as an aspect, and then inserted into the system by the *weaver* at proper places (by means of the

weaving process). In this manner, the implementation of the concern remains separated from other concerns.

1.1 Problem Statement

It is inevitable that some concerns crosscut a sizeable application, resulting in code scattering and tangling. This issue is particularly severe for security-related concerns: it is difficult to be confident about the security of an application when the implementation of its security-related concerns is scattered all over the code and tangled with other concerns, making global reasoning about security precarious. In this dissertation, we consider the case of access control, which turns out to be a crosscutting concern with a non-modular implementation in modern runtime environments such as the JVM [39] and the CLR [14].

The problems caused by crosscutting concerns in the implementation of software are well known. In the particular case of access control we can mention at least three specific issues:

- it is not easy to change the current access control implementation (*e.g.* to change the kind of security policies being enforced), because it is not modularly defined;
- programs that do not take security into account cannot be made security-aware without directly modifying them [88];
- forgetting to trigger access control checks at sensitive points in an application can lead to hard-to-spot security holes. If permission checks are not modularized, it is hard to reason about them globally.

1.2 Thesis

> Access control can be *fully* modularized as an aspect, leaving the programming language semantics completely *oblivious* to the presence of access control. Additionally, malicious code, including other aspects, will be *unable to interfere* with the access control aspect.

Here, we are concerned not only with the triggering of access control checks. We express

the *whole* access control infrastructure as an aspect, including the support for advanced features, namely privileged execution and capturable permission contexts. This work also describes a general-purpose aspect model that can be used to add access control to languages that do not include any (or very limited) support for it.

1.3 How to Read this Dissertation

Chapter 2 presents the necessary background concepts: access control (Section 2.1), aspect-oriented programming (Section 2.2), and scoping strategies (Section 2.3); ending with a summary of the current approaches to modular access control (Section 2.4).

The core of this dissertation is divided into three parts. The first part, composed by Chapter 3, presents our proposal for the modularization of access control: ModAC. We describe how ModAC, through the use of dynamically-deployed aspects and scoping strategies makes it possible to modularly define access control. The second part describes how we instantiated ModAC in a practical library for JavaScript. First, in Chapter 4 we describe AspectScript, our aspect-oriented language for JavaScript and then, in Chapter 5 we describe how we used AspectScript to implement ZAC, a practical library for access control. Finally, the third part, Chapter 6, formally proves that the access control defined in ModAC is secure and cannot be circumvented, even in the presence of other malicious aspects (the proofs of these claims can be found in Appendix A).

Chapter 2

Access Control, Aspects and Scoping Strategies

This chapters presents the three topics upon which this thesis is based. First, in Section 2.1 we introduce access control, focusing in the concept of permissions (to access a resource), and also explaining why whitelisting policies (based on permissions) are equivalent to blacklisting policies (based on restrictions). Then, in Section 2.2 we briefly present the aspect-oriented programming paradigm, concentrating on the pointcut-advice model. Later, in Section 2.3 we present scoping strategies as a mean for controlling the scope of aspects. Our work develops on top of this two building blocks. Finally, Section 2.4 enumerates related research concerning the modularization of access control.

2.1 Access Control

In this section we only give a brief introduction to access control. We return to this topic in Section 3.2, which gives a comprehensive description of the Java access control architecture, including illustrative examples.

Access control [65] is a cornerstone of every security architecture: it is the component in charge of ensuring that sensitive resources are accessed only by the entities authorized to do so.

In modern runtime environments such as the JVM [39] and the CLR [14], access control architectures rely on a fine-grained specification based on permissions. Permissions represent the ability

to access and use a particular resource (*e.g.* a file) in a certain manner (*e.g.* read-only or read-write). Fine-grained access control in these architectures allows one to assign different sets of permissions to different entities. Furthermore, stack inspection [35] is used to dynamically examine if a sensitive operation can be performed or not. This is known as *basic permission checking*.

The following code shows the essence of how basic permission checking is invoked inside the java.io.File.delete method:

```
SecurityManager.checkPermission(
  new FilePermission(file, SecurityConstants.FILE_DELETE_ACTION)
);
//actual deletion goes here...
```

The call to SecurityManager.checkPermission ensures that all the classes currently participating in the stack of execution have the SecurityConstants.FILE_DELETE_ACTION permission. If not, an exception is thrown to abort the file deletion.

The Java access control architecture also includes two other mechanisms: *privileged execution* and *first-class permission contexts*. Privileged execution allows a trusted entity to take responsibility for a certain action. This makes it possible for untrusted entities to access sensitive resources—such as the screen—in a controlled manner. First-class permission contexts allow the programmer to capture the set of permissions at a certain point and restore it later on, for instance to incrementally perform a long task—such as classloading—in different threads safely.

While these three mechanisms together provide a very powerful access control system, they also introduce modularity issues. Indeed, using basic permission checking is a crosscutting concern: in order to trigger stack inspection, explicit calls to the access control architecture are necessary. As a consequence, code related to permission checking ends up scattered at each and every place where sensitive resources are accessed, tangled with other concerns. In addition to the crosscutting nature of the *use* of access control, the *implementation* of access control is itself non-modular in the sense that it does not only lie in standard libraries, but depends on native support from the runtime environment. For instance, the Java VM provides specific support for reifying the stack and permission contexts. This native support in the VM is specific to (and can only be used for) access control enforcement.

Whitelisting vs Blacklisting. Policies based on whitelisting specify what resources the entities in the system *can* access (*e.g.* user X can use the printer). Conversely, policies based on blacklisting specifies what resources *cannot* be accessed (*e.g.* user Y cannot modify system files). In general, whitelisting is considered a safer approach because access to resources can be granted gradually,

minimizing the risk of inadvertently granting access to unneeded sensitive resources. This is not the case of blacklisting where it is possible to forget to restrict access to a resource. However, and despite the fact that whitelisting is superior from a conceptual point of view, blacklisting-equivalent approaches are successfully used in practice. A compelling proof of this comes from the widespread use of Java and C#. While their access control architectures appear to be based on whitelisting (because one declares permissions, not restrictions), they are in practice equivalent to blacklisting approaches. This stems from the fact that permission checking in these architectures has to be *explicitly* triggered at each and every relevant place in the code (by means of a call to SecurityManager.checkPermission(<Permission>) in Java and <IPermission>.Demand() in C#).

This means that forgetting to add the permission check associated to a sensitive resource in Java/C# is just like forgetting to restrict the access to that resource in a restrictions-based architecture. The dependency on explicit checks implies that the set of permissions is known in advance, and therefore, the set of restrictions can be calculated as the complement of the permissions. This is what makes the architectures of Java and C# equivalent, in practice, to a blacklisting architecture.

2.2 Aspect-Oriented Programming

Aspect-oriented programming makes it possible to modularize crosscutting concerns, *i.e.* concerns that does not fit into the initial view of the system, and in consequence end up scattered in several modules, and its corresponding code tangled with the code of other modules.

Every aspect-oriented language follows a *join point model* [53]. A join point model is comprised of three elements:

- Join points: the points of reference an aspect can affect;
- A means of *identifying* join points;
- A means of *effecting* execution at join points.

There are two kinds of join points: *dynamic* join points, that represent actions that occur at runtime, such as method calls or property accesses; and *lexical* join points, that represent locations in code, such as method declarations or annotations.

With respect to the means of identifying and effecting execution at join points, the predominant approach is the pointcut-advice (PA) model [53, 86]. In the PA model, identifying join points is

done by means of a pointcut, and effecting them is done by means of an advice. An aspect is a module that encompasses a number of pointcuts and pieces of advice.

Following is the shape of an aspect in AspectJ [49], which follows the PA model:

```
aspect FileDeletionAspect {
  pointcut fileDeletion(): execution(* File.delete()) && this(f);

  before(File f): resourceAccess(f) {
    SecurityManager.checkPermission(
      new FilePermission(this.path, SecurityConstants.FILE_DELETE_ACTION)
    );
  }}
```

The aspect FileDeletionAspect declares one pointcut and one piece of advice. The pointcut fileDeletion selects the join points of interest, in this case, the execution of the File.delete method. Typically, selection of join points is done using some kind of pattern matching (*e.g.* call(void Point.set*()) for all calls to setters in Point). Pointcuts can also expose contextual information such as the target of a call (target(..)) or, like in the example, the currently executing object (this(..)).

The piece of advice (before(): ...) declares that *before* each join point of interest, the specified action should be performed. In this case, we trigger an access control check before the file deletion. A piece of advice can also be declared as after or around to execute them after or instead of matching join points, respectively. In the latter case, the piece of advice can use the construct proceed() to execute the original behavior associated to the join point, if so desired.

Aspect Deployment & Scope. The way to "apply" an aspect to a system is to deploy it. There are two kinds of deployment: (a) *static* deployment (before execution time), and (b) *dynamic* deployment. (during execution time). The alternative used to deploy an aspect generally condition its scope. The "scope" of an aspect is the set of join points that the aspect potentially matches, *i.e.* against which its pointcuts are evaluated.

Static deployment implies global scope for the aspect *i.e.* it potentially sees all the join points of the program execution. Global scope makes it necessary for the pointcuts of aspects to include additional conditions to avoid unintentional matching of join points. Although possible, this results in unnecessarily complex pointcut definitions. For this reason, a number of languages supports dynamic deployment. For instance, CaesarJ [7] and AspectScheme [26] supports per-thread aspect deployment:

```
deploy(asp){ block }
```

Where the aspect asp sees all the join points generated in the dynamic extent of block. The aspect therefore has dynamic scope.

A variant of dynamic deployment is per-object deployment, also present in CaesarJ:

```
deploy(asp, obj);
```

In this case, the aspect asp sees all the join points occurring *lexically* within all methods of obj.

2.3 Scoping Strategies

Scoping strategies permit fine-grained control over the scoping semantics of a deployed aspect [72, 73]. A scoping strategy itself is specified by three *functions*:

- a *call stack* propagation function c specifies how an aspect propagates along with method calls;
- a *delayed evaluation* function d specifies whether or not an aspect is "captured" in objects when they are created;
- a *filter* function f specifies whether the pointcut of an aspect should be evaluated for a certain join point.

Intuitively, the c function allows controlling dynamic scoping of aspects, stopping propagation when a certain condition is met. The d function allows aspects to follow certain objects. Finally, the f function allows to filter out certain join points, acting as a local-refinement of the aspect pointcut. The *filter* function f is not used in this work, so we omit it from the discussion hereafter.

Propagation functions are themselves pointcuts, *i.e.* they are predicates over join points: c matches *call* join points for which the aspect should propagate, while d matches *object creation* join points. Figure 2.1 depicts the propagation of an aspect asp deployed with the following scoping strategy[1]:

```
[call(* Task.run()), target(Task)]
```

If asp is currently deployed (signaled by the gray rounded square labeled asp) , it propagates on calls to run (jp_{run}) but not on calls to clean (jp_{clean}). Therefore asp sees join points occurring during

[1] We use the concrete syntax [c,d] for deployment strategies, where c and d are defined similarly to AspectJ pointcuts.

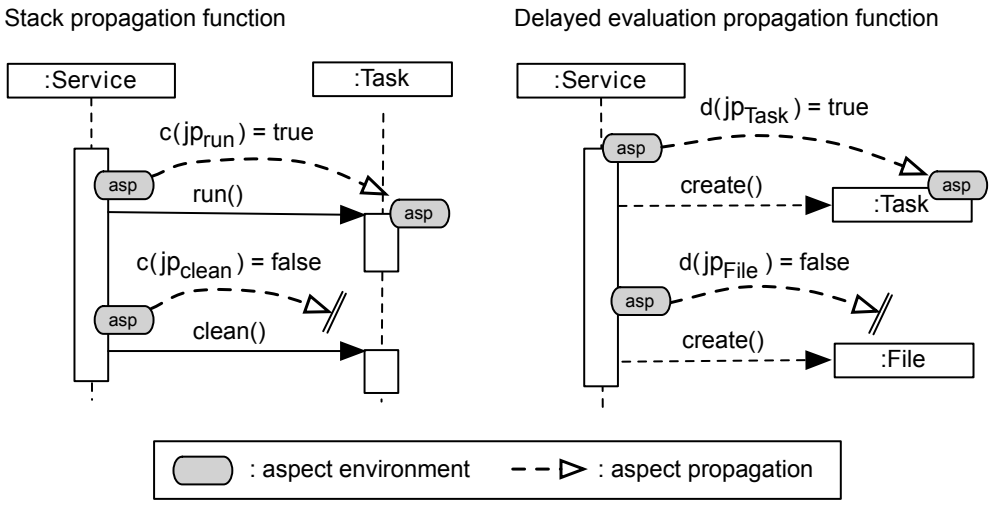

Figure 2.1: Propagation of aspects with scoping strategies

the execution of run. Similarly, it gets captured in new Task objects (jp_{Task}), and not in new File objects (jp_{File}). This means that it sees the subsequent activity of these Task objects.

Many examples of scoping strategies have been formulated elsewhere, for both local aspects [72] and distributed aspects [75], as well as for variable bindings [73].

2.4 Existing Approaches to Modular Access Control

The relation between aspects and security has a long history. A particular line of work that is complementary to access control as considered in this dissertation is the notion of data flow pointcuts and properties. With respect to access control, there are related proposals in the area of modularization of access control checks through AOP and through other alternatives, whose relation to AOP is not explicitly established. We describe all of them in the remainder of this section.

Data-Flow Aspect-Oriented Approaches to Security. Most aspect-oriented approaches targeted to security concerns other than access control are based on data flow. Pointcuts for identifying returned values and their later usage are proposed in [52], and their formal semantics is presented in [4]. These kinds of pointcuts are used to ensure the correct usage of values, according to the constraints of the entities that returned the values and the entities that finally use them. An improved version of the same concept, but working over the control-flow graph of applications is presented in [57]. With these pointcuts, it is not only possible to reason about returned values, but also about any identifiable event (join point) in the flow of a program.

AOP-based Modularization of Access Control Checks. Several proposals have been presented to modularize access control based on aspect-orientation. However, these approaches only factor out access control checks into aspects. None of them tackles the issue of modularizing advanced features of access control such as privileged execution and permission contexts. Each approach tackles the modularization of access control checks to a different degree. In [24,70,89,90], the focus is just to factor out particular access control checks into aspects (similarly to the FileDeletionAspect aspect in Section 2.2). Conversely, in [42, 83, 91], the focus is to build an access control library of reusable aspects. The idea is to provide abstract aspects triggering access control at certain points yet to be defined by concrete implementations of these aspects. This approach makes it easier to add new points where access control must be checked by reusing the existing aspect definitions. Finally, a proposal that goes a step further is [56], where a security pattern for access control is defined. This pattern is designed to enforce a correct handling of access control and must be followed each time a sensitive resource is about to be accessed. As in the case of access control libraries, the pattern is implemented as an aspect.

Non AOP-based Modularization of Access Control Checks. In [28], *inlined reference monitors* are used to maintain access control state in the application. The approach is very close to AOP (even though the connection is not explicitly established) in that actions (*e.g.*PERFORM SECURITY UPDATE <action >) are performed whenever certain events occur (WHEN <method signature >). Just like in the AOP-based modularization of access control checks, this proposal only factors out access control checks.

The summary of related proposals presented in this section is by no means exhaustive. Each one of the following chapters have its own dedicated related work section. We follow this structure because at this point we have not presented all the elements necessary to understand all the related work.

Chapter 3

ModAC: Aspectizing Access Control

This chapter presents our solution for the modularization of the Java access control architecture[1]. We first show a solution based on AspectJ, the most popular aspect-oriented extension to Java, that must rely on a separate automata infrastructure. We then put forward a novel solution via dynamic deployment of aspects and scoping strategies. Both solutions, apart from providing a modular specification of access control, make it possible to easily express other useful policies. This is the first step towards a modularized access control.

3.1 Introduction

The security architecture of Java consists of several components [37]: data typing, memory management, bytecode verification, secure classloading, cryptography, secure communications, public key infrastructure, and access control. Among them, Java access control (hereafter JAC) is a cornerstone of the platform: it is in charge of ensuring that sensitive resources (*e.g.* the filesystem) are accessed only by the classes authorized to do so.

More than a decade ago, the JAC architecture evolved from a very limited sandbox model into a more flexible model based on stack inspection [38]. In the former model, classes are considered as either trusted or untrusted. This condition determines which classes are able to access sensitive resources and which classes are not. In the latter model, finer-grained control based on permissions is introduced. Permissions represent the ability to access and use a particular resource (*e.g.* a file) in

[1]This chapter is based on the publication: "Aspectizing Java access control" [79].

a certain manner (*e.g.* read-only or read-write). Fine-grained access control in the JAC architecture allows one to assign different sets of permissions to different classes. Furthermore, *stack inspection* is used to *dynamically* examine if a sensitive operation can be performed or not.

The three main mechanisms of the JAC architecture based on stack inspection are:

- **Basic permission checking.** When a sensitive resource is about to be accessed, a call to the JAC API triggers a stack inspection algorithm, which *checks* whether all the classes in the current stack of execution possess the necessary permission to access the resource. If not, an exception is thrown. This basic behavior prevents the confused deputy problem [41] from happening, *i.e.* an untrusted class cannot lead a trusted one to access (or modify) a sensitive resource on its behalf.

- **Privileged execution.** In some scenarios, it is necessary for a class to access a sensitive resource on behalf of another –possibly untrusted– class. For this, the JAC architecture supports *privileged execution*.

- **Permission contexts.** When accessing a sensitive resource, it can be necessary for a class to use the permissions present at another point in the execution of the application. The JAC architecture provides the means to capture a *permission context* and restore it later on.

While these three mechanisms together provide a very powerful access control system, the JAC architecture suffers from modularity issues. Indeed, in order to trigger permission checking through stack inspection, an *explicit* call to the JAC architecture is necessary. As previously explained in Chapter 2, this is known as a crosscutting concern.

Figure 3.1 shows all classes of the standard distribution of Java where access control is necessary[1]. The bars represent classes and the different lengths depict the size of the class in lines of code. The black parts inside the bars point out the actual places where basic permission checking is performed.

In addition to the non-modularity of permission checking, the implementation of stack inspection is itself problematic: while part of it resides in the Java libraries (the stack inspection algorithm), it also relies on support from the Java Virtual Machine implementation for snapshotting the current stack of execution (permission context capturing). Moreover, this native support is specific to (and can only be used for) access control enforcement.

[1] Figure obtained by using XRef. http://www.eclipse.org/ajdt/xref.

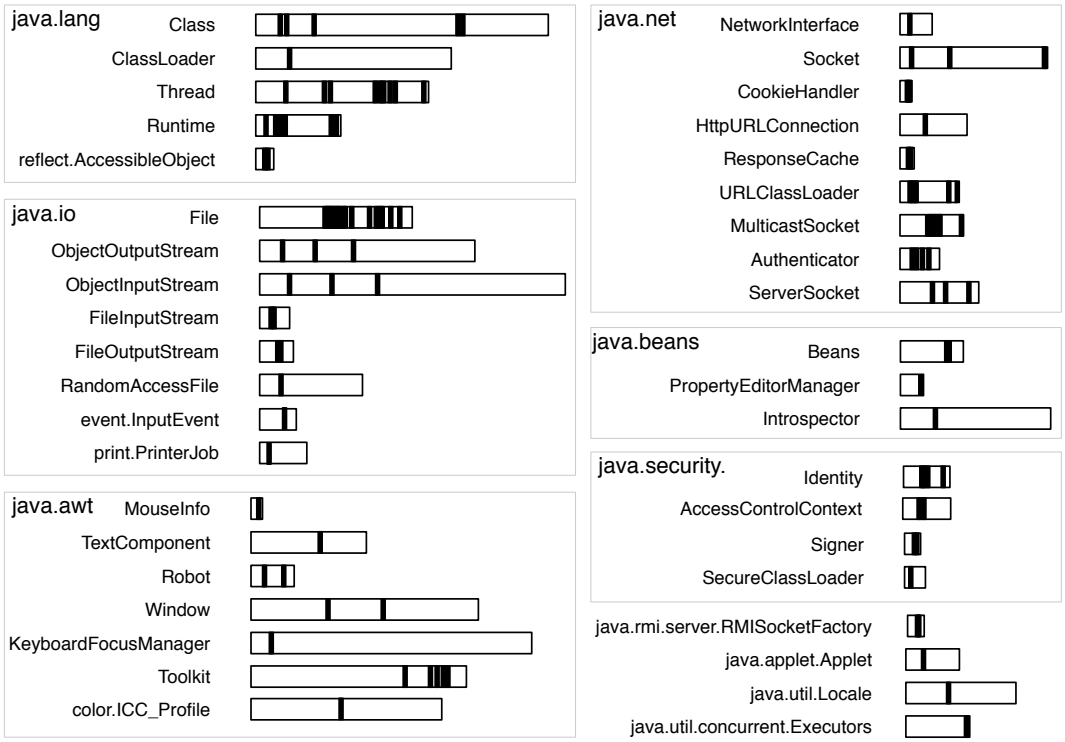

Figure 3.1: Access control checks in the standard distribution of Java 1.5.

This chapter describes how to define the JAC architecture entirely within AspectJ, the *de facto* standard aspect-oriented extension of Java [49]. Although we succeeded in this attempt, the result was not completely satisfactory. While basic permission checking can be defined in terms of restriction aspects using the scoping mechanisms of AspectJ, privileged execution and permission contexts require a specialized access control automaton. For this reason, we used the state-of-the-art proposal in terms of scope control: *scoping strategies* [72, 75]. This time, the solution fitted completely within the aspect language, providing several interesting advantages:

a) it is extremely *succinct and elegant*: the central piece, the AccessControlAspect, is only 10 lines of code.

b) it *fully supports* the three main mechanisms of the JAC architecture: basic permission checking, privileged execution, and permission contexts.

c) it is *more expressive*[1] than the JAC architecture: it supports per-object permissions (rather than just per-class), as well as more dynamic permission assignment policies.

[1]Following [30], we say that a language *A* is more *expressive* than another language *B*, when translating a program with occurrences of a feature of *A* to *B* requires a global reorganization of the program.

d) it does not rely on specialized runtime mechanisms for stack inspection, but rather relies on a *general-purpose* aspect scoping construct that can be used in other domains.

The rest of this chapter is organized as follows. Section 3.2 presents and illustrates the use of the JAC architecture through a simple running example. Section 3.3 explores the use of AspectJ, highlighting why a specialized automaton for access control is necessary. Section 3.4 shows how scoping strategies can be used to simply express the full JAC architecture, in a finer-grained and dynamic manner. Section 3.5 reflects on the advantages and future challenges derived from an aspectized access control architecture. Section 3.6 compares the solutions based on AspectJ and scoping strategies in detail. Section 3.7 discusses related work and Section 3.8 summarizes.

3.2 Access Control in Java

In this section we first describe the JAC architecture, and then illustrate its implementation by means of an incrementally-refined example.

3.2.1 Problem Statement

The JAC architecture enforces a discretionary access control policy [65] in which granting access to sensitive resources depends on the identity of the requester. In this regard, the JAC architecture [37] is based on several elements, namely: protection domains, classes, permissions, and threads. A *protection domain* is defined as a set of *classes* that are loaded (by a classloader) from a certain codebase (represented by a URL). Each protection domain is granted a set of *permissions*. Through the protection domain they belong to, classes share the same set of permissions. In the JAC architecture, the smallest entity a certain permission can be granted to is a protection domain. Permissions are granted (by default) to protection domains in a separate Java policy file[1]. This is done before the application starts by the administrator of the system. In the remainder of this thesis, we refer to "the permissions of the protection domain of a class" simply as "the permissions of the class" for conciseness.

The JAC architecture defines a *system* protection domain. By default, only classes in this domain are considered trusted and hence are allowed to access sensitive resources. Examples of resources accessible by the system protection domain are the filesystem, network, screen, mouse,

[1] A description of this file and its syntax can be found in §3.3 of [69].

and keyboard. Currently, all code shipped as part of Java belongs to the system protection domain (§4.1 [69]).

The last elements in the JAC architecture are *threads*. It is fundamental to consider them in the equation because a thread of execution can cross different protection domains. For instance, when user code wants to print some text to the standard output, the thread of execution goes from a user protection domain[1] to the system protection domain. The inverse case is also possible when a class in the system protection domain invokes a method pertaining to a class in a user protection domain (this is the case of callbacks in the Swing GUI library for instance).

To summarize, let us consider the access to a resource associated to a permission P. A permission context is a reification[2] of a stack of execution, denoted $[C_1m_1, \ldots, C_nm_n]$, where the stack grows to the right, and C_im_i denotes method m_i of class C_i. In the the JAC architecture, the resource access is considered secure in the current permission context if and only if one of the following conditions holds[3]:

1. If no C_im_i started a privileged execution, then all C_im_i must possess P.

2. If C_pm_p started the last privileged execution and no permission context was specified, then all C_im_i for $p \leq i \leq n$ must possess P.

3. If C_pm_p started the last privileged execution and a permission context S was specified, then all C_im_i for $p \leq i \leq n$ must possess P; and S itself must comply with conditions 1, 2 or 3 for P.

There are several possible implementations to enforce such conditions. In the remaining of this section we present the stack-based implementation of the JAC architecture. Then, we detail two aspect-oriented implementations: one based on AspectJ (Section 3.3), and one based on per-object deployment and scoping strategies (Section 3.4).

[1] We use the term "user protection domain" to refer to a protection domain other than the system protection domain, potentially untrusted.

[2] Reification is the process by which the state of the interpreter is passed to the program itself, suitably packaged (reified) so that the program can manipulate it [36].

[3] More detailed and formal descriptions of the the JAC architecture can be found elsewhere [37, 38, 48, 69].

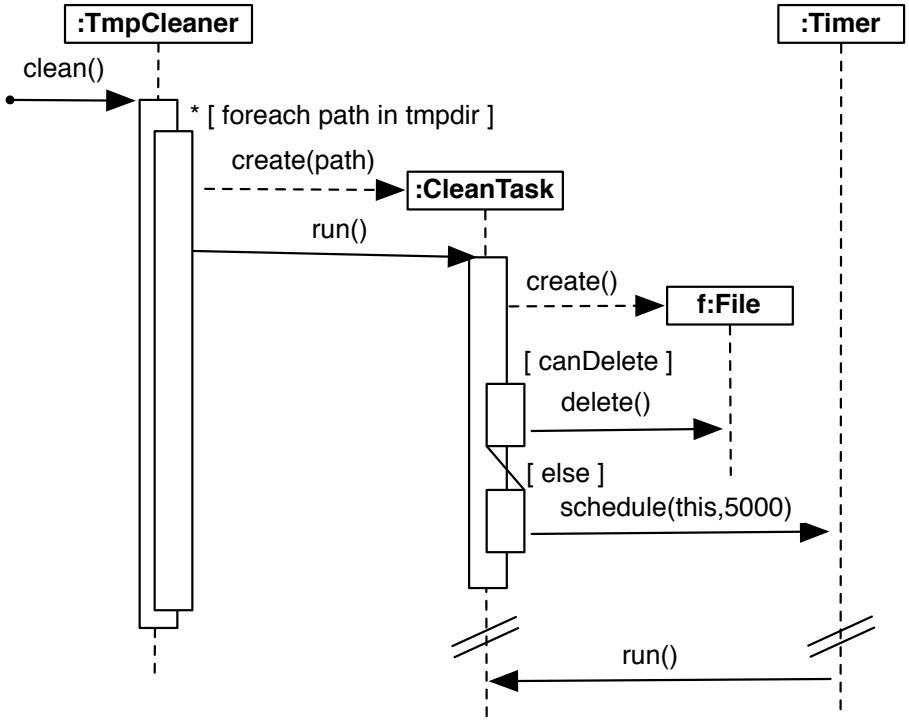

Figure 3.2: Illustrative Example: a service for cleaning a temporary directory.

3.2.2 Illustrative Example

Consider a service whose function is to periodically clean a temporary directory. This kind of service could be useful, for instance, to applications that require great amounts of disk space or to disk-cleaning applications.

Figure 3.2 depicts the architecture of the service: TmpCleaner is the entry point for client applications through its clean method. Each file in the temporary directory is deleted by an auxiliary CleanTask instance, which encapsulates the deletion of a particular file. The service is designed this way because if the file cannot be immediately deleted, the CleanTask instance schedules itself for later execution with a certain delay using a preexistent java.util.Timer. When the delay is over, the Timer invokes the CleanTask.run method again (the asynchronous nature of this call is denoted by the double diagonal lines at the bottom of the figure). If the file still cannot be deleted, the CleanTask schedules itself again until the deletion can be finally performed. This service exercises the three mechanisms of JAC presented in Section 3.1:

- **Basic permission checking.** As a first approach, we can ensure that TmpCleaner can only be used by clients that have permissions to write to the temporary directory.

- **Privileged execution.** If we rather want to allow TmpCleaner to be invoked by any class (this is safe considering the nature of the temporary directory, and makes the service usable in a wider range of contexts), the deletion of files must be performed as a privileged execution.

- **Permission contexts.** As explained before, some deletions are postponed by scheduling CleanTask instances on a Timer. They are subsequently executed, but in a completely different context by the timer thread. This means that the permission context present at the original call to TmpCleaner.clean must be *captured* and later on *reinstalled* when the CleanTask instances are scheduled by the timer.

3.2.3 Basic Permission Checking

In the JAC architecture, before accessing a system-sensitive resource, the class accessing the resource must explicitly perform a call to SecurityManager.checkPermission. This method takes as a parameter a java.security.Permission instance representing the permission to be checked. For example, here is the essence of the code in the java.io.File.delete method:

```
SecurityManager.checkPermission(
   new FilePermission(this.path, SecurityConstants.FILE_DELETE_ACTION)
);
//actual deletion goes here...
```

When checkPermission is invoked, the permissions of *all* classes in the stack of execution of the current thread are inspected. If one of these classes is not granted the permission to delete the file, a java.security.AccessControlException is thrown to cancel the whole operation. Following is the pseudo-code for this stack inspection mechanism[1] (m is the index of the top frame in the stack):

```
i = m;
while (i > 0) {
   if (class i does not have the permission)
      throw new AccessControlException();
   i = i − 1;
}
```

This basic access control mechanism is very useful in the most common scenario: a class in the system protection domain is being used by a class in a user protection domain. This is the situation

[1] This algorithm and the remaining ones in this section are described in detail in the JAC architecture specification, §4 of [69]. The only difference is that in the documentation, the stack inspection algorithm continues in the stack that was active when the current thread was created (§4.3). In our case, we abstract from this implementation detail and assume a one-piece stack.

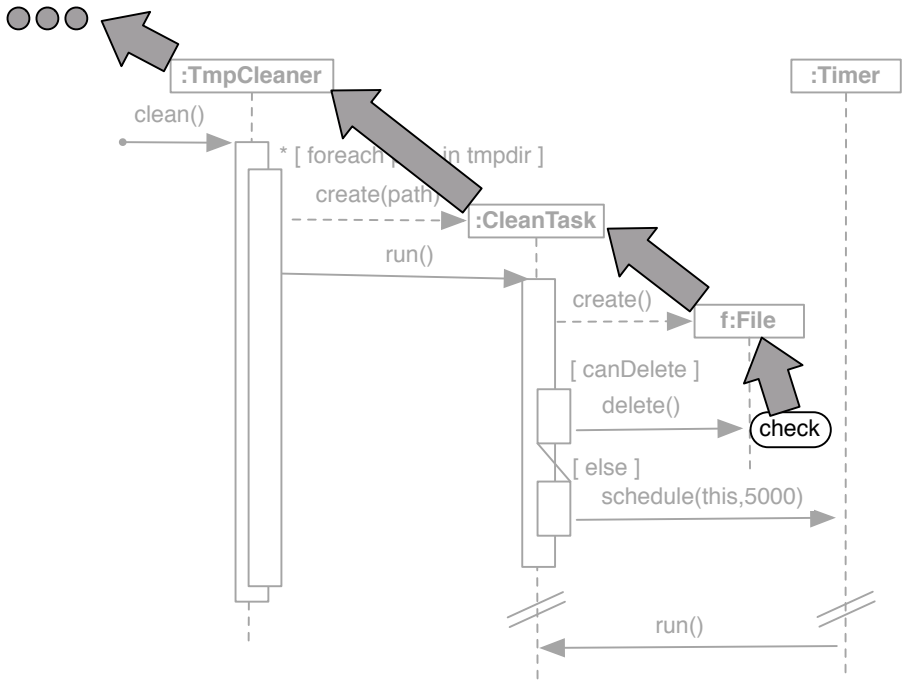

Figure 3.3: Basic permission checking algorithm in the JAC architecture.

in the running example: TmpCleaner service is not part of the system protection domain. Hence, the file deletion should be allowed only if CleanTask (the class that calls File.delete) and the other classes in the stack, *all* have been granted the corresponding permission (recall that the system protection domain is granted all permissions).

The path followed by the stack inspection algorithm can be observed in Figure 3.3. The arrows show the order in which the classes are inspected, starting from File, the class that triggers the basic permission checking algorithm (the circled "check" in the figure). The arrow pointing outwards of TmpCleaner denotes that its clients are also inspected.

3.2.4 Privileged Execution

In some cases, there is the necessity for a class to access a sensitive resource on behalf of another class. In the example, it is safe to let any class use the TmpCleaner service because only the service determines which files to delete; it cannot be told to delete other, non-temporary files (assuming that the service itself is granted access only to the temporary directory).

To allow this, the JAC architecture provides a way to perform a privileged action[1]. The semantics of privileged execution is that during the execution of the action, the permissions to consider must exclude the ones of classes in the stack above the class that started the action, but must include the ones of that class. The updated stack inspection algorithm that takes into account the execution of privileged code is:

```
i = m;
while (i > 0) {
  if (class i does not have the permission)
    throw new AccessControlException();
  if (class i started a privileged action)
    return;
  i = i − 1;
}
```

The algorithm inspects the stack class by class, just like the previous algorithm, but when it sees a class that started a privileged action, it returns normally (having checked the permission for that class before). Note that this feature does not intrinsically imply a security breach because the permissions of the entity that starts a privileged execution are maintained.

In our example, TmpCleaner should execute its complete body inside a privileged action. This way, any client will be able to use the service, regardless of its own permissions. Following is how TmpCleaner specifies which code is executed as a privileged action:

```
class TmpCleaner {
  ...
  public void clean() {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        for(String path: tmpdir){
          new CleanTask(path).run();
        }
      }});
  }}
```

The doPrivileged method starts a privileged action. The action to perform in this privileged context is specified as the body of the run method of a java.security.PrivilegedAction instance.

Figure 3.4 shows the sequence diagram for the previous code. The class that started a privileged action is signaled with a star (TmpCleaner), acting as a barrier through which no arrow can go.

[1]The word "privileged" is misleading in this context because no privilege is necessary to start a privileged action: it can be started by any class, trusted or not (§4.2.2 of [69]).

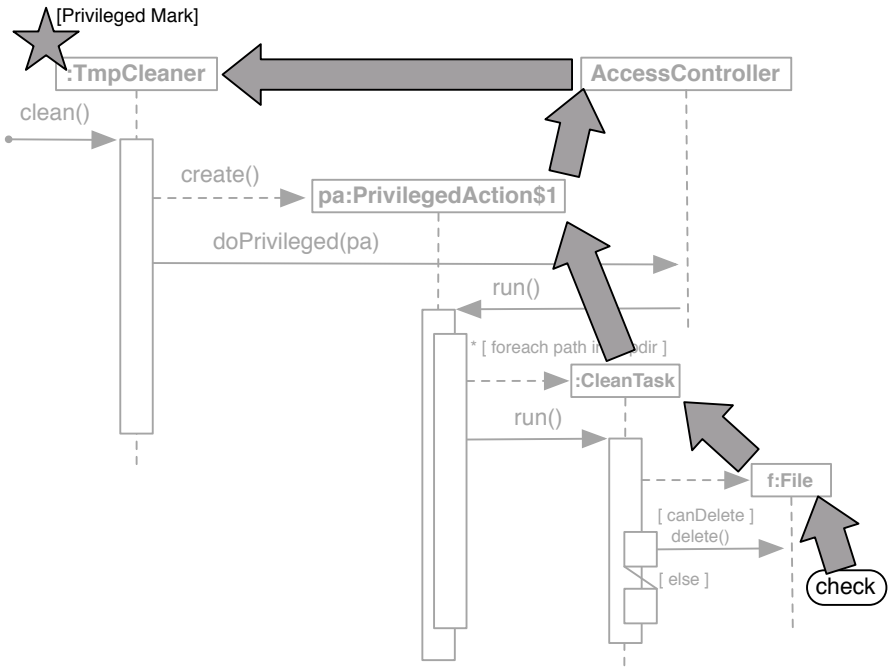

Figure 3.4: Permission checking with privileged execution.

This barrier prevents the stack inspection algorithm from inspecting further classes in the stack. Executing the body of the clean method in a privileged action allows any class to use the service, no matter if it cannot write the temporary directory.

3.2.5 Permission Contexts

Despite the privileged action inside TmpCleaner.clean, there is a second path for basic permission checking not shown in Figure 3.3. This path goes from CleanTask to Timer. It is a different path because the call is asynchronous and hence, it is not in the control flow of TmpCleaner.clean. For this reason, the file deletion is not under the privileged action, so stack inspection goes over all the stack. Therefore, our intent of using the permissions of TmpCleaner when deleting files is not complete: the permission context of the original call to TmpCleaner.clean must be *captured* and used when deleting the files later on.

The use of a permission context ensures that the classes inspected in an indirect file deletion are the same than in a direct deletion. This is necessary to guarantee that changes in the permissions granted to these classes are correctly reflected in the execution.

In the JAC architecture, AccessController.getContext() is used to capture the current permission

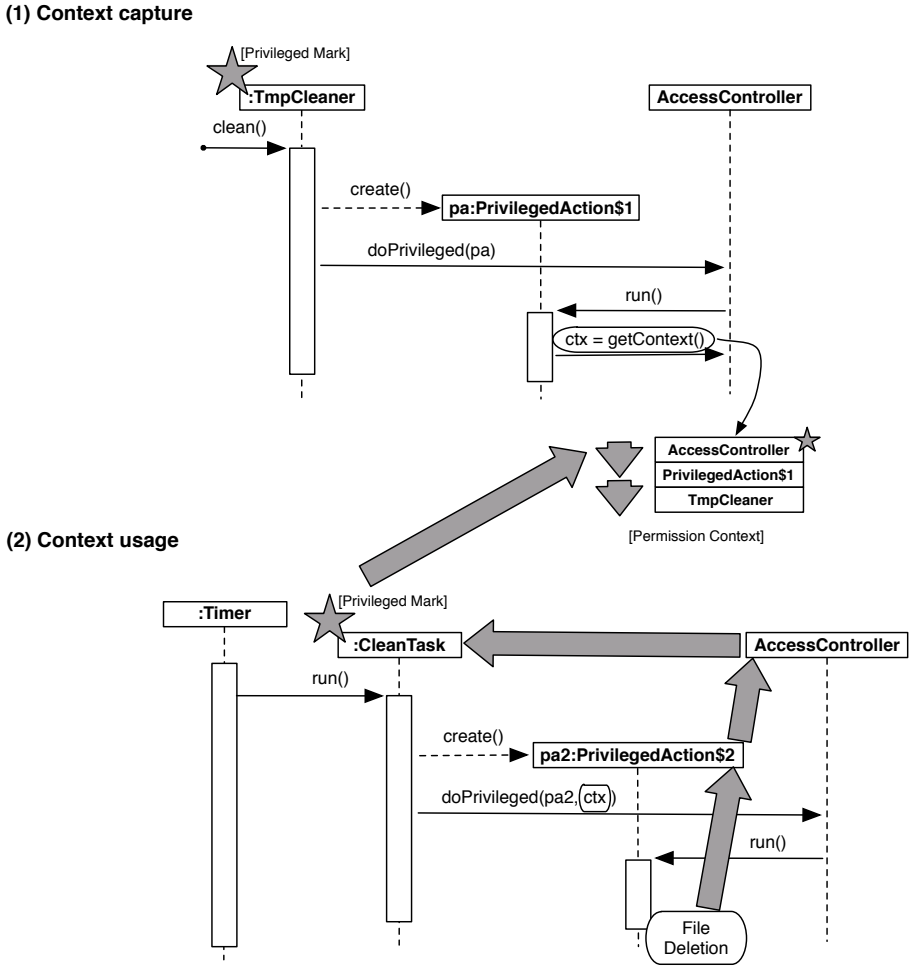

Figure 3.5: Permission checking with permission context capture (1) and further usage (2).

context. We use it at the beginning of the privileged action in the clean method. The context object is essentially a reification of the current execution stack. Later, the CleanTask.run method runs the deletion of the file as a privileged action, passing the previously-captured context as an extra parameter (we modified the constructor of CleanTask to specify the context). This ensures that the deletion is done in the permission context corresponding to the TmpCleaner class: if TmpCleaner can delete files in the temporary directory, the deletion proceeds; if it cannot, an access control exception is thrown.

The code below illustrates the capture of the permission context in TmpCleaner.clean, and its subsequent use in CleanTask.run:

```
//TmpCleaner (context capture)
public void clean() {
  AccessController.doPrivileged(new PrivilegedAction() {
    public Object run() {
```

```
        AccessControlContext ctx = AccessController.getContext();
        for(String path: tmpdir){
          new CleanTask(path, ctx).run(); //pass ctx
        }
      }});
}

//CleanTask (context usage)
public void run() {
  if(canDelete()) {
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        file.delete();
      }}, this.ctx); //use ctx to delete the file
  }
  else{ /* self−scheduling on timer */ }
}
```

Following is the complete stack inspection algorithm for permission checking in Java, updated to support the use of permission contexts:

```
i = m;
while (i > 0) {
  if (class i does not have the permission)
    throw new AccessControlException()
  if (class i started a privileged action)
    if (a context was specified)
      continue inspection on context
    return;
  i = i − 1;
}
```

The semantics of passing a permission context to a doPrivileged call is that once the class that initiated the privileged action is reached, the stack inspection continues in the stack the permission context represents. If no context was supplied, the algorithm returns normally, just as in the previous case for handling privileges.

Figure 3.5 depicts the updated scenario. First, the permission context at the beginning of the clean method is represented as the stack of execution at that point (the involved classes are exactly the ones of Figure 3.4). Then, the doPrivileged call in CleanTask.run passes a reference to this permission context. When deleting a file, the captured permission context is used. This completely eliminates the second path from CleanTask to Timer.

3.3 Java Access Control with AspectJ

In this section we describe how to implement the three main mechanisms of the JAC architecture using AspectJ [49], the *de facto* aspect-oriented extension to Java.

3.3.1 Factoring out Access-Control Checks

Through the pointcut-advice model (PA model from now on), it is straightforward to modularize access control checks. It is only necessary to define an aspect whose pointcut identifies accesses to sensitive resources, and whose advice triggers stack inspection:

```
aspect FileDeletionPermission {
  pointcut resourceAccess(File file):
      execution(boolean File.delete()) && target(file);

  before(File file): resourceAccess(file) {
    SecurityManager.checkPermission(
      new FilePermission(file.path, FILE_DELETE_ACTION)
    );
  }}
```

Deploying this aspect is equivalent to explicitly invoking SecurityManager.checkPermission in File.delete (Section 3.2.3). However, the fundamental advantage of the aspect-oriented approach is that explicit calls to SecurityManager.checkPermission are not necessary anymore. As long as one aspect per kind of permission is defined, the calls end up modularized in these aspects.

This approach is possible because the AspectJ pointcut language is expressive enough to identify all points where access control is necessary. Although further research is required for the general case, we have conducted a study for the standard Java distribution, which shows that it is possible to specify pointcuts for all occurrences[1]. This by-product of our study constitutes, to the best of our knowledge, an original contribution.

[1]The complete list can be found at http://pleiad.cl/research/3sec

3.3.2 Stack Inspection Semantics via Scope Control

Most prior proposals for the modularization of access control in Java uses the aforementioned approach for factoring out access-control checks (see Section 2.4). However, this way of aspectizing access control still relies on the non-modularized and specialized stack inspection implementation of Java. It is interesting to explore if aspects can allow for a fully modular JAC architecture. To do so, we need to express the stack inspection semantics via another mechanism.

Before we go into details on how to implement stack inspection in a modular way, let us describe the kind of aspects we will need to use for the task.

Aspects like FileDeletionPermission simply delegate access control decisions to the default stack inspection algorithm. This algorithm ensures that all classes in the stack have the corresponding permission when a resource is about to be accessed. Consequently, we call these aspects *permission aspects*. However, since the default implementation of stack inspection cannot be part of a modularized solution (because it is not modularly defined), we need another kind of aspects based on a different mechanism for access control enforcement: *restriction aspects*. A restriction aspect, instead of invoking SecurityManager.checkPermission in its advice, throws an exception as soon as it *sees* the resource access its pointcut identifies:

```
aspect FileDeletionRestriction {
  pointcut resourceAccess(File file):
      execution(boolean File.delete()) && target(file);

  before(File file): resourceAccess(file) {
    if(file.path.equals("...")) {
      throw new AccessControlException();
    }
  }}
```

Because a restriction aspect throws an exception when its pointcut matches a join point, its *scope* must be limited so that it only sees illegal resource accesses. In other words, its scope must be controlled. In the next section we explore how the scoping features of AspectJ can be used for this purpose.

3.3.3 Aspectizing Basic Permission Checking

Determining whether a resource access is legal or not can be considered a matter of *flow of execution*: if the access is in the control flow of a class with a restriction to access the resource, then the access is illegal. Otherwise, the access is legal.

Aspect languages usually provide means to reason about control flow: aspects can be limited to see only the join points that occur within certain flows of execution. AspectJ, in particular, provides two options in this regard. First, AspectJ features percflow deployment, which creates and deploys a new aspect instance for each flow of execution starting in certain join points (specified using a pointcut). Second, AspectJ also features a cflow pointcut, which matches join points only under a particular flow of execution. A cflow pointcut can be conjuncted with other pointcuts as an extra condition over the join points to select. Using these AspectJ control flow features to limit the scope of restriction aspects, implementing basic permission checking is straightforward. Following is the FileDeletionRestriction aspect, updated so that it only sees join points in the control flow of the execution of any method in UntrustedClient (additional restrictions are necessary for other untrusted entities). The code includes two specification options: *(a)* using percflow and *(b)* using cflow. Only one is necessary[1].

```
aspect FileDeletionRestriction
        percflow(execution(* UntrustedClient.*())) (a) {

  pointcut resourceAccess(File file):
      execution(boolean File.delete()) && target(file);

  before(File file): resourceAccess(file) &&
                        cflow(execution(* UntrustedClient.*())) (b) {
    if(file.path.equals("...")) {
      throw new AccessControlException();
    }
  }}
```

[1] cflow only controls the *scope* of the aspect, while percflow also controls its *instantiation strategy*: each time a method of UntrustedClient is executed, a new aspect instance is created (if not already in the control flow of another UntrustedClient method execution).

3.3.4 Privileged Execution

The previous implementation of basic permission checking using either percflow or cflow is simple and direct. However, it cannot be used for realizing privileged execution. Once in a privileged execution, restriction aspects must not see a resource access even if it occurs in the control flow of a method of an untrusted class. In the case of percflow, once a restriction aspect has been deployed, it cannot be undeployed for the extent of a privileged execution, which is exactly what is needed. The case for cflow is similar. This problem is even more evident if we consider multiple privileged executions, initiated by different classes, with a mixture of trusted and untrusted classes involved in the stack.

It turns out that the scoping mechanisms offered by AspectJ do not suffice to express the JAC architecture stack inspection semantics. For this reason, it becomes necessary to manually manage state upon which to decide whether a resource access is legal or not. In the following, we detail a solution that maintain access control state through a *pushdown automaton*.

3.3.5 Access Control Automaton

In the following, we maintain access control state through a *pushdown automaton*. This approach corresponds, in essence, to various proposals for stack inspection alternatives [28, 67, 84, 85]. The novelty is that in this work we use an aspect-oriented approach for updating the automaton. With the use of this *access control automaton*, restriction aspects are deployed with *global scope* and must check the automaton to decide when to throw an exception.

Each access control automaton[1] has two logical states: LEGAL and ILLEGAL, representing legal and illegal resource accesses, respectively. For a given automaton, it is in state LEGAL when all classes in the stack of execution have the permission to access the resource it guards. It is also in state LEGAL when a privileged execution (started by a trusted class) is in progress. If none of these conditions hold, the automaton is in state ILLEGAL. Figure 3.6 depicts the access control automaton for privileged execution[2]. This automaton is updated each time a class enters and exits the stack, and also when entering and exiting a privileged action. The transition to follow depends on whether the class is subject to the restriction associated to the automaton or not; in the former case the class is considered untrusted, and in the latter case, trusted. Transitions are prefixed with enter/exit and

[1] We consider one access control automaton per restriction aspect for convenience and practical reasons: a global automaton would have an exponential number of states: 2^n, where n is the total number of restrictions.

[2] We tested this automaton with JFlap. http://www.jflap.org.

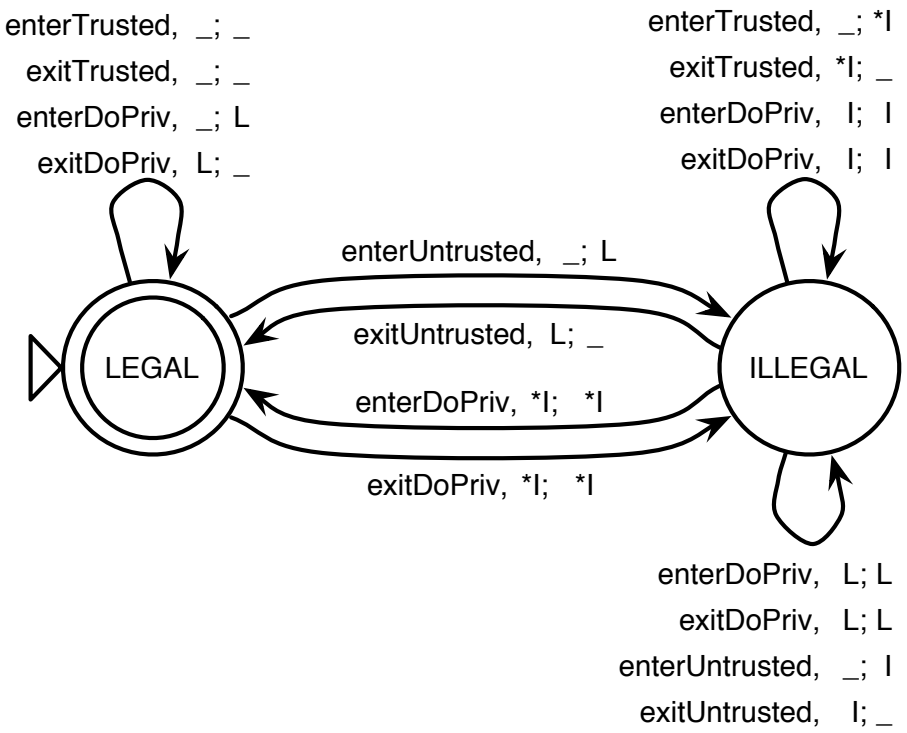

Figure 3.6: Pushdown automaton for privileged execution. Transitions are defined by the notation: <*event*>,<*value on top of the stack*>;<*value pushed onto the stack*>.

suffixed Trusted/Untrusted/DoPriv for this purpose. The stack of the access control automata is used to remember from which state an enter* transition originated, so that the automaton can correctly switch back to the corresponding state on an exit* event. The symbols in the stack can be: L, I, and *I; for "coming from LEGAL", "coming from ILLEGAL", and "coming from ILLEGAL, but the last class was trusted" respectively. The *I symbol is useful to distinguish a privileged execution started by a trusted class from one started by an untrusted class. The _ symbol is used to ignore what is on the top of the stack (when specified in the second position of transitions), and to prevent a push operation from occurring (when specified in the third position of transitions).

The implementation of the FileDeletionRestriction aspect supporting privileged execution, using and updating its access control automaton is presented in Figure 3.7. The top part of the figure shows the pointcut and advice of the restriction aspect. The only variation there is the check to determine if the automaton is in an ILLEGAL state, in which case an exception is thrown. The bottom part of the figure shows the code for updating the automaton. The untrustedMethods pointcut selects the set of methods belonging to classes that have the corresponding restriction. Each time one of those methods is executed, the automaton is updated accordingly. The same process is performed for trusted classes (the methods not selected by untrustedMethods), and when doPrivileged

```
aspect FileDeletionRestriction {
  //resource access pointcut and advice
  pointcut resourceAccess(File file): ... ;
  before(File file): resourceAccess(file) {
    if(!automaton.inLegalState() /* scope check */ && file.path.equals("...")) {
        throw new AccessControlException();
      } }

  //automaton handling
  private static InheritableThreadLocal<AC_Automaton>
    automaton = new InheritableThreadLocal<AC_Automaton>() {
      protected AC_Automaton initialValue() {
        return new AC_Automaton();
      }
      protected AC_Automaton childValue(AC_Automaton parent) {
          return new AC_Automaton(parent);
      } };
  pointcut untrustedMethods(): ... ;
  Object around(): untrustedMethods() {
    automaton.get().enterUntrusted();
    try { return proceed(); }
    finally { automaton.get().exitUntrusted(); }
  }
  Object around(): !untrustedMethods() { //trusted
    automaton.get().enterTrusted();
    try { return proceed(); }
    finally { automaton.get().exitTrusted(); }
  }
  Object around(): call(* AccessController.doPrivileged(..)) {
    automaton.get().enterDoPriv();
    try { return proceed(); }
    finally { automaton.get().exitDoPriv(); }
  }}
```

Figure 3.7: FileDeletionRestriction and its access control automaton

is executed. The try/finally blocks are necessary to update the automaton even when proceed throws an exception. Finally, the automaton is accessed through an InheritableThreadLocal variable, which allows each automaton to correctly reflect the access control state of the corresponding thread.

By using these updated restriction aspects, the JAC architecture semantics for privileged execution can be implemented. However, the solution combines AspectJ constructs and a specialized access control automata. Actually, each access control automaton rebuilds its own view of the execution stack.

```
public class PermContext {
  public static Map <Class, AC_Automaton> getContext() {
    Map <Class, AC_Automaton> context = new HashMap <Class, AC_Automaton>();

    context.put(FileDeletionRestriction.class, FileDeletionRestriction.getContext());
    // put automaton instances for other restrictions
    ...
    return context;
  }
}
```

Figure 3.8: Permission context implemented as a map.

3.3.6 Permission Contexts

In order to support permission contexts, the access control automata can be extended with the necessary infrastructure for snapshotting and reinstalling a permission context.

On the JAC side, a permission context is represented by the set of classes in the current thread of execution. This is so because the stack inspection algorithm needs the execution stack to tell whether a resource access is legal or not. On the access control pushdown automaton side, the current states and stacks of the automata always instantaneously distinguishes between a legal and an illegal resource access. Therefore, the permission context at an execution point can be considered as the set of access-control-automaton instances at that point.

In implementation terms, a permission context maps restriction aspects and automaton instances. As shown in Figure 3.8, the getContext method of the PermContext class can be used to get this map. It uses the getAutomaton method provided by each restriction aspect to get a copy of the current automaton instance.

When a piece of code is going to be executed under a given context, each restriction aspect needs to change its automaton instance by the respective instance in the given context. The current automaton instance needs to be saved in order to be restored after the piece of code is executed. Since contextual execution can be nested, the automaton instances need to be stored in a stack. The FileDeletionRestriction aspect of Figure 3.7 is extended to support permission contexts as Figure 3.9 shows. First, the aspect is equipped with the getAutomaton method returning a copy of the current automaton instance. Second, a new piece of advice is defined that intercepts a contextual execution and changes the current automaton instance by the respective automaton instance given in the passed context. Third, the aspect is provided with a stack variable implemented as an

```
aspect FileDeletionRestriction {
  /* code of Figure 3.7 */

  Object around(PrivilegedAction action, PermContext context):
            call(* AccessController.doPrivileged(..,..)) &&
            args(action,context){

    stack.get().push(automaton.get());
    automaton.set(context.get(getClass()));
    try { return proceed(action,context); }
    finally { automaton.set(stack.get().pop()); }
  }

  public static AC_Automaton getContext() {
    return new AC_Automaton(automaton.get());
  }

  private static InheritableThreadLocal<Stack<AC_Automaton>>
    stack = new InheritableThreadLocal<Stack<AC_Automaton>>(){
      protected Stack<AC_Automaton> initialValue(){
        return new Stack<AC_Automaton>();
      }

      protected Stack<AC_Automaton> childValue(Stack<AC_Automaton> parent) {
        Stack<AC_Automaton> stack = new Stack<AC_Automaton>();
        stack.addAll(parent);
        return stack;
      }
    };
}
```

Figure 3.9: FileDeletionRestriction and its access control automaton with support for permission contexts.

InheritableThreadLocal object, which is used to keep the automaton instances that need to be restored after contextual executions.

3.3.7 Summary

The implementation presented in this section constitutes, to the best of our knowledge, the first modular realization of the JAC architecture entirely based on aspect orientation. However, the solution is not completely satisfactory. While basic permission checking can be defined in terms of restriction aspects using the scoping mechanisms of the AspectJ language, privileged execution and permission contexts require these mechanisms to be replaced with a specialized access control automaton.

The reason for falling back to an implementation based on access control automata is that AspectJ only supports a limited notion of scoping of aspects. First, it is impossible to undeploy an aspect once it has been deployed using percflow, which is necessary for privileged execution. Second, using cflow, it is not possible to distinguish a legal resource access from an illegal one once in the control flow of multiple privileged executions.

3.4 ModAC: Modular Access Control

In this section we show how scoping strategies [72, 73] can be used to define the JAC architecture in an aspect-oriented manner by controlling the scope of restriction aspects. This contrasts with the AspectJ solution, which is based on *global* deployment of restriction aspects and explicit scope control through access control automata.

We first explain how a simple scoping strategy (see Section 2.3) allows an elegant specification of the three main mechanisms of the JAC architecture (basic permission checking, privileged execution, and permission contexts). We then describe in details why such a simple definition effectively meets the requirements.

3.4.1 The Scoping Strategy for Java Access Control

The specification of the JAC architecture by means of scoping strategies is succinct and elegant. Figure 3.10 presents the AccessControlAspect aspect, in charge of deploying restriction aspects. The underlying idea is to deploy restriction aspects on objects with a proper scoping strategy, so that the relevant parts of the activity of the objects is under control of the restriction aspects.

Deploying restriction aspects. The AccessControlAspect aspect deploys restriction aspects *on objects* when they are created. First, the init pointcut matches all object creations (line 3), binding the newly-created object to the instance parameter (line 4). The associated advice (lines 13-16) deploys the corresponding restriction aspects on that object, using deployOn (line 16). The set of restriction aspects that corresponds to a particular object is determined by the getRestrictionsFor method (line 14). This method encapsulates the binding of objects to their protection domain. One possible implementation is to mimic the JAC architecture by returning the restriction aspects that correspond to the permissions declared in the Java policy file. Another implementation would be to return

```
1  aspect AccessControlAspect {
2    //creation of objects
3    pointcut init(Object instance): execution(*.new(..))
4                                        && this(instance);
5
6    //access control scoping strategy
7    ScopingStrategy acss = [
8      !(call(* Object.doPrivileged(..)) &&
9          if(jp.getTarget() == jp.getThis())),
10     target(AccessControlContext)
11   ];
12
13   before(Object instance): init(instance) {
14     Aspect[] restrictions = getRestrictionsFor(instance);
15     //per−instance deployment
16     deployOn[acss](restrictions){ instance };
17   }}
```

Figure 3.10: Access control aspect for deploying restriction aspects.

restrictions based on dynamic conditions, such as the kind of user currently interacting with the application, as in role-based access control [32].

Access control scoping strategy. Restriction aspects are deployed on objects (line 16) with the *access control scoping strategy* defined in lines 7-11. The call stack propagation function of the strategy expresses both basic permission checking and privileged execution, as will be explained in Section 3.4.2 and 3.4.3, respectively. Essentially, it specifies that a restriction aspect always propagates on the call stack, except on privileged calls. The delayed evaluation propagation function expresses the capture of permission contexts. It ensures that restriction aspects propagate to AccessControlContext instances; therefore, creating such an object is a means to take a snapshot of the restriction aspects present at that point in time. This is explained in Section 3.4.4.

3.4.2 Basic Permission Checking

Basic permission checking dictates that accesses to system-sensitive resources must be denied if a class on the stack does not have the corresponding permission. The call stack propagation function of the access control scoping strategy (lines 8-9 of Figure 3.10), in the absence of privileged execution (*i.e.* no calls to the doPrivileged method), always evaluates to true (line 9 is explained in Section 3.4.3). In consequence, the restriction aspects of the objects participating in the stack unconditionally propagate through all method invocations: they see all actions that are about to be performed, and can deny them as appropriate.

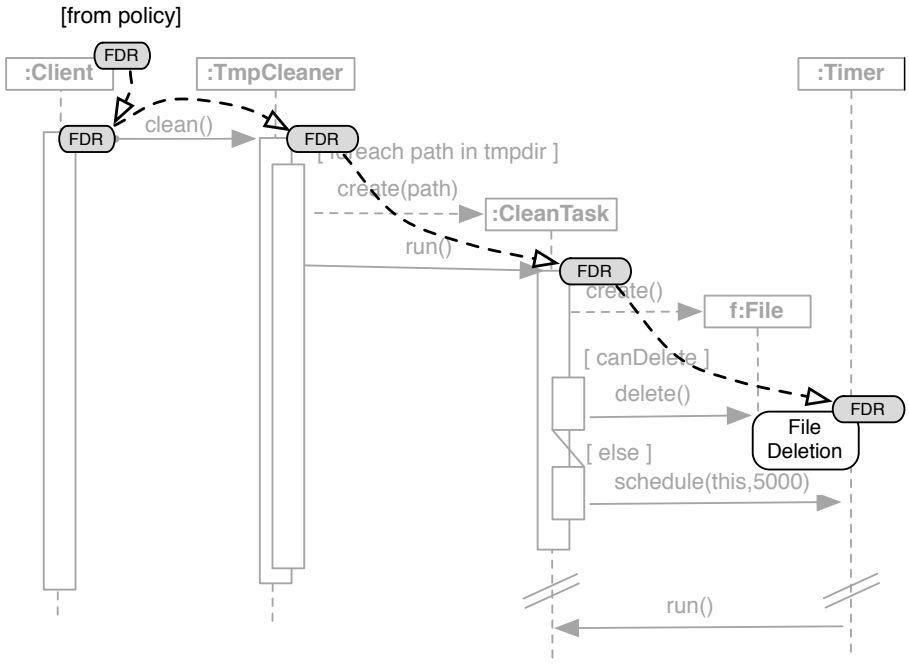

Figure 3.11: Restriction aspects propagation for basic permission checking

Figure 3.11 shows how the FileDeletionRestriction (FDR) aspect is propagated on method invocations in the TmpCleaner service. For the sake of clarity, we consider FDR to be the unique restriction aspect in the system. The only untrusted object, with FDR deployed on it, is the client that invokes TmpCleaner.clean; all remaining instances[1] are considered trusted. The figure shows how FDR always propagates from the initial call to TmpCleaner.clean until the actual file deletion, following the semantics of the stack propagation function.

3.4.3 Privileged Execution

Recall from Section 3.2.4 that only the restrictions associated with the entity that initiates a privileged action and the subsequent ones must be enforced, while those associated with entities higher in the call stack are ignored.

In terms of scoping strategy for restriction aspects, this means that restriction aspects must not see resource accesses in the dynamic extent of the privileged action: they should not propagate when a privileged action starts executing. However, the restriction aspects associated with the entity initiating the privileged action must be present during the execution of the action.

[1]We assume that all method calls are performed over an object. Static methods still fit the model if we convert them into instance methods or consider classes as objects.

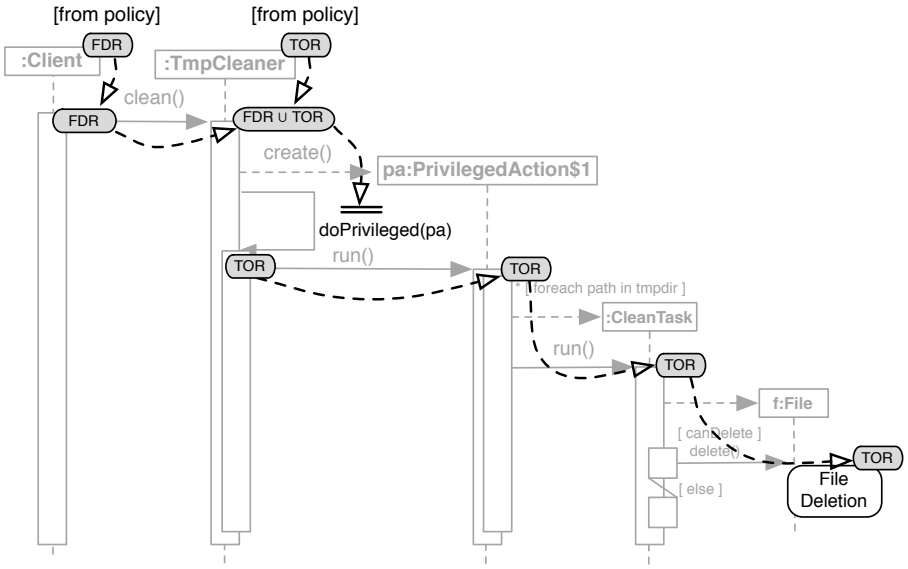

Figure 3.12: Restriction aspects propagation for permission checking with privileged execution

We introduce a special doPrivileged method for privileged code execution, similar to the doPrivileged static method of the AccessController class in the JAC architecture. The semantics is equivalent except that this method is now defined on Object, as an *instance* method:

```
class Object {
  ...
  public final Object doPrivileged(PrivilegedAction action){
    return action.run();
  }}
```

In order to support the requirements of privileged execution stated above, doPrivileged is only effective on *self calls*. When called on this—and only in that case—the current restriction aspects do not propagate. This is specified by the call stack propagation function of the access control scoping strategy shown on Figure 3.10, line 8-9: propagation on the call stack stops upon calls to Object.doPrivileged only if jp.getThis() == jp.getTarget() (*i.e.* it is a self call). Note that because doPrivileged is an instance method of the object that is initiating the privileged action, restrictions associated to that object (as specified by the policy file) *do* propagate.

To illustrate this point better, let us extend our scenario by assuming that the policy file specifies that TmpCleaner is restricted to writing to the tmp directory by means of a "Temp Only Restriction" (TOR aspect) deployed on it. Figure 3.12 shows how the FileDeletionRestriction (FDR) aspect is propagated through the call to the clean method. Then, both FDR and TOR are stopped when performing a self call to doPrivileged (since the call stack propagation function of the access control scoping

strategy evaluates to true in this case). The resurgence of TOR in the execution of doPrivileged is simply because it is deployed on TmpCleaner. Eventually, the file deletion is allowed, because only TOR is present.

3.4.4 Permission Contexts

Recall from Section 3.2.5 that using permission contexts involves two processes: context capture and context usage. Capturing the current permission context is performed by invoking the AccessController.getContext() method, and reinstalling it is performed by starting a privileged execution with the context as a parameter. The semantics of using a context for a privileged execution stipulates that, in addition to the restrictions of the entity starting the privileged action (as in the previous section), the restrictions enforced also include the restrictions of the permission context.

In terms of the scoping strategy for restriction aspects, this means that it should be possible to capture the restrictions present at some point in time, associate them with a permission context, and later on use them in a privileged execution. Let us first detail how permission contexts are captured and then, how to reinstall them.

Capturing restrictions in permission contexts is handled by the scoping strategy: the delayed evaluation propagation function permits restriction aspects to propagate to all newly-created instances of the AccessControlContext class (Figure 3.10, line 10). In consequence, creating such an object captures in that object all the current restriction aspects, *i.e.*. the current permission context:

```
AccessControlContext ctx = new AccessControlContext();
```

In order to reinstall the restriction aspects propagated to an AccessControlContext instance, we introduce an overloaded version of the doPrivileged method, taking a context as a second parameter, just like in the original JAC architecture:

```
public final Object doPrivileged(PrivilegedAction action, AccessControlContext context){
  return context.run(action);
}
```

The method calls the run method on the context object to include it in the stack, thereby ensuring that all captured restriction aspects are reinstalled, and consequently propagate on the stack. The AccessControlContext class is simply defined as follows:

```
public final class AccessControlContext{
  public final Object run(PrivilegedAction action){
```

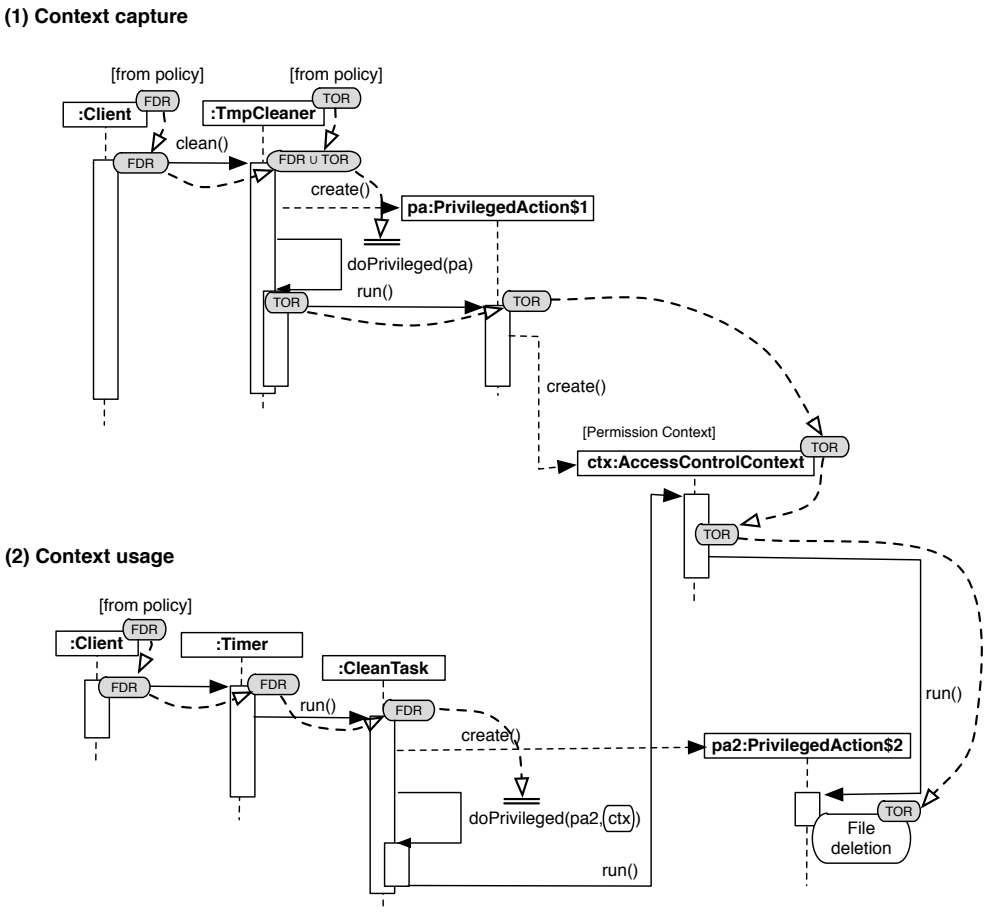

Figure 3.13: Restriction aspects propagation for permission context capture (1) and further usage (2).

```
    return action.run();
}}
```

Note that for the same reasons exposed in Section 3.4.3, the restriction aspects of the object that invokes doPrivileged are present in the doPrivileged body. Therefore the adequate set of restrictions (the union of the restrictions in the object initiating the privileged action and the restrictions in the context object) is reestablished.

To illustrate this process, let us again extend our scenario by considering that the entity that starts the timer also has a FileDeletionRestriction. Figure 3.13 shows how the FDR and TOR aspects are propagated in method invocations in the TmpCleaner service. Both aspects are stopped at doPrivileged calls, but TOR is captured on an AccessControlContext instance that is later on used by CleanTask to start a privileged action. This results in the inclusion of TOR in the stack again. At the point of the file deletion, the only restriction aspect is TOR, so the deletion is eventually allowed.

3.5 Evaluation of Aspectized Access Control

In this section we discuss some of the advantages and disadvantages of the aspect-oriented implementations of the JAC architecture presented in Sections 3.3 and 3.4. We consider five viewpoints in this analysis, namely default behavior, modularity, security, expressiveness, and management.

3.5.1 Default Behavior

As we demonstrate in Chapter 6, the use of ModAC correctly and securely enforces an access control policy by means of restriction aspects and the access control scoping strategy. However, from a security perspective, its is important to clearly highlight which are the consequences of misusing the security constructs. In the following we describe and compare the JAC and the modularized architectures with respect to their behavior when one of the three access control mechanisms are not used correctly.

Basic Permission Checking. As explained in Section 2.1, forgetting to deploy a restriction aspect is equivalent to forgetting to invoke the stack inspection mechanism. In both cases, the same security flaw is introduced: the access control policy is not enforced, leaving the sensitive resource unprotected.

Privileged Execution. Failing to start a privileged execution leads to the same scenario in the JAC architecture as in the modularized architectures. In the former case, more classes than expected will be checked for permissions; and in the latter case, more restrictions will be enforced. In both cases the consequence is the same: an incorrect configuration is considered and therefore, the access control policy is incorrectly enforced (although in this scenario the access to the sensitive resource is more likely to fail, thus it does not necessarily represent a security hole).

Permission Contexts. Forgetting to reinstall a permission context has the same consequence in the JAC architecture as in the modularized architectures. In the former case, the current stack will be checked for permissions instead of the stack captured in the permission context; and in the latter case, the current set of restrictions will be enforced instead of the restrictions captured in the permission context. Both cases introduce the same security flaw: unauthorized entities will eventually be able to access the sensitive resource since the access control policy is being enforced considering an incorrect configuration.

3.5.2 Modularity

An aspect-oriented solution implies two major advantages in terms of modularity. First, access control is defined exclusively in *dedicated modules*: the restriction aspects. Consequently, there is no scattering and tangling of the access control concern. Second, looking at pointcut declarations should immediately reveal where access control is being enforced, without having to navigate the code to find the relevant points. In consequence, it is *difficult* to miss security-sensitive operations because they are specified in a single place.

Logically, the validity of this latter advantage depends on the complexity of the declaration of pointcuts, and on the expertise of the developer. However, according to the study we conducted to determine whether it was possible to specify AspectJ pointcuts for all occurrences of basic permission checks, we believe that for the majority of cases, the access to a sensitive resource is expressed by simple a pointcut definition. For instance, more than the 85% of the restriction aspects necessary to express access control modularly in Java follow this pattern:

```
aspect Restriction{
  pointcut access(): execution(<method or constructor>);
  before(): access(){
    if(<some condition>){
      throw AccessControlException();
    } }
}
```

Finally, one issue that appears is that without considering access control, applications are not prepared for the unexpected scenario of failing to access a resource. This is a common case experienced today in the JAC architecture since the java.security.AccessControlException class extends from RuntimeException, and therefore it is not mandatory to declare it in method signatures. The result of this at runtime is exceptions being thrown and not handled by the application. In the modularized architectures, this problem remains the same. However, thanks to the fact that every restriction is implemented by a particular restriction aspect (opposed to a permission-independent stack inspection algorithm), it is possible to not always throw an exception but to provide an alternate "safe" behavior (more on this in Section 5.3.4).

For the cases when throwing an exception is necessary, exception handling can be treated as another concern, and means have been proposed to deal with it modularly [16, 33].

3.5.3 Security

Turning checkPermission calls into restriction aspects prevents malicious aspects from interfering with access control. For example, consider the following aspect:

```
aspect MaliciousAspect {
  pointcut accessControl(): call(void SecurityManager.checkPermission(..));
  void around(): accessControl() {}
}
```

This aspect selects all calls to checkPermission and does nothing instead, thus completely annihilating access control. Restriction aspects solve this problem because they provide an implicit way of performing access control checks: having no explicit calls to checkPermission means no join points for malicious aspects to match on.

However, some aspect languages permit aspects to observe not only the computation of the base application, but also the computation of other aspects. For instance, AspectJ provides an adviceexecution pointcut to select advice execution. Using this pointcut, a malicious aspect can interfere with access control by skipping some or all advice executions. This means that advice of restriction aspects can be prevented from throwing exceptions on illegal resource accesses, thus eliminating access control.

A couple of tentative solutions to this issue, mainly based on further limiting the scope of aspects, can be devised. A first alternative is to use the notion of levels of execution, as in stratified aspects [13] and execution levels [74]. In a nutshell, the idea is to structure computation into levels, starting with base computation at level 0. Aspect computation is by default considered as occurring at level 1, and is therefore invisible to other aspects. If needed, aspects can be deployed at higher levels of execution, thereby possibly observing other aspects. In this model, restriction aspects can be deployed on a sufficiently-high level so that no other aspect can see (or interfere with) their activity. A second alternative is to classify aspects into user (untrusted) and kernel (trusted) categories, where user aspects cannot interfere with kernel aspects. For instance, join points derived from computation of kernel aspects can be hidden from user aspects. In chapter 5 we present a novel solution to this problem that makes it possible for all aspects to see all join points but not to interfere with access control.

3.5.4 Expressiveness

An aspectized access control architecture makes it possible to easily express useful policies, such as the Chinese wall policy [15], which is based on the history of resource accesses.

The aim of the Chinese wall policy is to avoid conflicts of interest in decision making. In the Chinese wall model, sensitive resources are called *objects* (*e.g.* a financial report); an object pertains to a *company dataset* (*e.g.* an oil company); and a company dataset pertains to a *conflict of interest class* (*e.g.* oil companies). The idea is that once an entity accesses a certain object, it must no longer be able to access other objects in a different company dataset within the same conflict of interest class (*e.g.* financial reports of two different oil companies). This way, the entity cannot combine information from one conflict of interest class to make strategic decisions.

The Chinese wall policy is known to be difficult to implement using the JAC architecture because "[..] the JVM would need not only to monitor object interaction but also keep a history of it" (§7.2 of [37]). However it can be directly defined using a restriction aspect:

```
aspect ChineseWallRestriction {
  List accessedCompanies = new ArrayList();

  pointcut objectAccess(Object object): /* accesses to objects */ && target(object);

  before(Object object): objectAccess(object) {
    Company company = companyOf(object);
    if(accessedCompanies.contains(company)){
      return; //OK if the company was already accessed
    }
    for(Company c: accessedCompanies){
      //accessing an object within the same COI class?
      if(coiOf(company) == coiOf(c)){
        throw new AccessControlException();
      }
    }
    accessedCompanies.add(company);
  }}
```

The key in this implementation is that aspects intrinsically monitor the execution of the system. This contrasts with the architecture based on explicit permission checking of Java. Note that implementing the Chinese wall policy as a restriction aspect makes it possible to use all the features associated to stack-based access control, including privileged execution and capturable permission

contexts. This is so because the ChineseWallRestriction aspect is deployed—as any other restriction aspect—with the access control scoping strategy.

3.5.5 Management

There are two tasks involved in the management of an access control architecture. First is the task of the developer or security auditor which is in charge of identifying sensitive resource accesses and take action to secure them. And second is the task of the policy administrator in charge of defining the access control policy that every actor in the system is subject to.

In order to secure a sensitive resource access in the JAC architecture, it is only necessary to insert an access control check to trigger the stack inspection algorithm for a certain permission. In the modularized architectures, the developer must define a restriction aspect whose pointcut identifies the resource access. In terms of the effort required to complete these tasks in each case, one could arguably state that defining an aspect is more complex than inserting an access control check. This is true until the developer gets used to define restriction aspects; as presented in Section 3.5.2, most restriction aspect follow the same, simple, pattern.

In terms of policy enforcement, the default way in the JAC architecture is to define a security policy file. In this file, the policy administrator statically defines the permissions for a protection domain (*i.e.* for the classes loaded by a certain classloader from a certain codebase). In the modularized architectures, emulating this static definition of the policy is direct. Actually, it is possible to use the same file format if desired.

Besides the static definition of a policy, in the JAC architecture it is also posible to define the permissions of a protection domain based on dynamic conditions. This is done by installing an instance of a subclass of java.security.Policy, which is in charge of assigning permissions to protection domains by overriding the getPermissions(ProtectionDomain) method. In the ModAC architecture, the equivalent method is getRestrictionsFor(Object) in Figure 3.10. Again, the same semantics can be obtained if the getRestrictionsFor only takes into account the protection domain of the object.

Consequently, and taking into account both the static and dynamic definition of the access control policy, the only difference in terms of management for the policy administrator is when enforcing restrictions at the object level is necessary. For this case, an expressive means to refer to objects is necessary in order to be able to meaningfully assign them restrictions. We leave this issue as a topic for future research.

3.6 Access Control Automata vs Scoping Strategies for Access Control

In this section we present a comparison between access control using AspectJ and access control using scoping strategies. Although a solution based on AspectJ is possible in order to implement the JAC architecture, the use of scoping strategies brings many benefits, from both a conceptual and a practical viewpoint.

3.6.1 General Purpose vs Domain Specific

In terms of the infrastructure necessary to support a modularized JAC architecture, both solutions are similar. For the AspectJ solution, a per-thread automaton is necessary to keep track of the current access control state of the stack given the restrictions of entities participating in it. For the scoping strategies solution, a per-thread aspect environment is required to maintain the set of restriction aspects currently in scope. However, a fundamental conceptual difference between both approaches is whether they can serve purposes other than access control. On the one hand, the automata infrastructure is specific and therefore, only useful for this problem. On the other hand, the aspect environments infrastructure is general-purpose and can be used for much more than access control. Examples where dynamically-deployed aspects with proper scope are useful are manifold [72, 75]. Actually, scoping strategies are generally applicable, including to other kinds of adaptations beyond aspects [73].

3.6.2 Expressiveness

In this section we compare both solutions in terms of expressiveness. By expressiveness, we refer to the fact that a feature that is directly expressible with dynamic deployment and scoping strategies such as per-object restrictions requires additional data structures and several modifications to be implemented in AspectJ. The reason for this, as detailed before in Section 3.3, is the limited scoping control in the language.

3.6.2.1 Increased Dynamism

Dynamic aspect deployment allows greater control over where and when an aspect affects a program [72]. While the AspectJ solution is limited to static deployment and refining the scope of aspects necessarily implies modifying the access control automata, the scoping strategies solution can take advantage of dynamic deployment. In particular, dynamically deploying AccessControlAspect introduces the possibility to activate and deactivate access control at will, depending on runtime conditions in the application.

In the original JAC architecture, all classes are subject to access control. Even classes in the system protection domain are checked by the stack inspection algorithm! The reason is that access control is global, it cannot be deactivated[1]. Analogously, in the AspectJ implementation, restriction aspects are active even when the automaton state is legal. Again, this is due to the global scope of restriction aspects in that implementation.

Conversely, in the scoping strategies implementation of the JAC architecture, activating (resp. deactivating) access control is just a matter of deploying (resp. undeploying) AccessControlAspect[2]. In consequence, through dynamic deployment developers can specify in which execution flows of the program restriction aspects are deployed on objects, and hence, which objects are subject to access control. Examples of scenarios where the conditional activation/deactivation of access control can be useful are:

- **Unconditional Activation**. When transferring control flow to methods on unknown, dynamically-loaded classes: execution of these methods is always untrusted, therefore, AccessControlAspect should be unconditionally deployed.

- **Conditional Activation**. When loading an untrusted site in a browser tab: only dynamic content (applets, flash animations, JavaScript code, etc.) in that specific tab is untrusted, therefore, AccessControlAspect should be deployed only in case an untrusted site is about to be loaded. This per-page security enforcement is what NoScript[3] does.

- **Unconditional Deactivation**. When a user marks an application as trusted: all code in that application is now considered trusted, therefore, AccessControlAspect should be uncondition-

[1] A call to doPrivileged does not deactivate access control. It only puts a mark denoting that stack inspection must stop there.

[2] All objects which already have restriction aspects deployed on them are still subject to access control.

[3] NoScript (http://noscript.net) is a Firefox extension that allows JavaScript, Java, Flash and other dynamic content plugins to be executed only by trusted Web sites chosen by the user. It has more than 2 million users.

ally undeployed in future executions.

- **Conditional Deactivation**. When executing commands in a command-line interpreter: certain commands do not access sensible resources, like echo and date (only the screen, which the user already has access to at this point), therefore, AccessControlAspect can be undeployed for them.

These are only some direct advantages derived from dynamic deployment for access control. Further study may reveal more, such as a policy that dynamically enforces different security levels (with different performance trade-offs) depending on the state of the application.

3.6.2.2 Increased Granularity

In the original JAC architecture and in the AspectJ modularized implementation, access control is specified at the class level. However, access control at the level of classes has limits in terms of expressiveness [34, 60]. Let us borrow an example (not the solution) from [34] to illustrate how per-object access control permits the expression of finer-grained policies. In a hospital management system, users can be doctors or patients. Doctors can both read and update records of their patients, whereas patients can only read their own records. If the system is modeled with Doctor and Patient classes, assigning them the corresponding read and write permissions, a security breach is immediately introduced. Firstly, *all* doctors are able to read and write *all* patient records, not only the records of their *own* patients. At the same time, patients are able to read *other* patient records, probably including private information. Conversely, with object-based access control, one direct solution is just to deploy the following restriction aspects on Doctor and Patient instances:

```
aspect DoctorsRestriction {
  pointcut patientAccess(Patient patient):
      execution(* Patient.*(..)) && this(patient);

  before(Patient patient): patientAccess(patient) {
    Doctor doctor = ...; //reference to the doctor this aspect is deployed on
    if(!doctor.treats(patient)){ //this doctor does not treat the patient
      throw new AccessControlException();
    }
  }}

aspect PatientsRestriction {
  pointcut readRecord(Patient patient):
      execution(Record Patient.getRecord()) && this(patient);
```

```
before(Patient patient): readRecord(patient) {
  Patient thisPatient = ...; //reference to the patient this aspect is deployed on
  if(thisPatient != patient){ //the record is from another patient
    throw new AccessControlException();
  }
}}
```

With these restriction aspects deployed on Doctor and Patient instances, doctors are not able to read or update records of other patients; and patients are not able to read records of other patients.

To illustrate another feature derived from the increased granularity, let us extend the example by adding a second kind of doctor, a doctor-in-chief, that is allowed to view and modify the records of patients of other doctors (*i.e.* no restriction is deployed on it). Now suppose that a doctor wants to pass a patient record to another doctor, but s/he does not want the record to be modified. Since the doctor receiving the record could be a doctor-in-chief, it is not safe to just pass the record.

To solve this issue, the doctor can, defensively, deploy the following restriction on the record to prevent it from being modified:

```
aspect RecordRestriction{
  pointcut writeRecord(Record record): execution(* Record.set*(..)) && target(record);

  before(Record record): writeRecord(record){
    Record thisRecord = ...; //reference to the record this aspect is deployed on
    if(thisRecord == record){
      throw new AccessControlException();
    }
  }}
```

Notice that this restriction is deployed on the Record object itself, not in the entities potentially accessing it. Even if the doctor that receives the record shares it with other entities, they will not be able to alter it, regardless of their own restrictions.

3.6.2.3 Increased Control

Scoping strategies can be used to define other kind of policies by easily changing the scope of restriction aspects. For instance, one can define a pervasive policy. This policy ensures that all actions *derived* from the execution of a restricted object must be restricted as well. By derived we

refer to the actions directly generated by the object and the actions generated by other objects it creates. This policy is implemented by deploying restriction aspects with the *pervasive* scoping strategy [72] defined as [true,true]. Actions directly generated by the object are subject to the policy because the call stack propagation function is always true, so restriction aspects always propagate on the stack. Actions indirectly generated by the object are also subject to the policy because the delayed evaluation function is always true, so restriction aspects always propagate to new objects. Consider the following example:

```
//(1) code in an untrusted object
File file = new File("/etc/passwd");
trustedObject.file = file;

//(2) code in trustedObject
this.file.delete();
```

Under stack inspection semantics, the file deletion would be allowed because there is no trace of the untrusted object in the stack. Conversely, in a pervasive policy semantics, the call to delete throws an exception. The reason is that when the untrusted object creates the File object, the restriction aspects of the former propagate to the latter, due to the delayed evaluation function. This can be seen as a kind of history-based access control [1].

Implementing this policy using the solution based on AspectJ and automata involves (besides adding per-object restrictions support) a complete rewrite of the access control automata. Conversely, as shown above, using scoping strategies is just a matter of changing the propagation functions.

3.6.3 Optimization Opportunities

The solution based on scoping strategies presents more direct optimization opportunities than the solution based on access control automata.

First, because access control automata are not defined within AspectJ, they cannot benefit from the optimizations made to the compiler/runtime of the language. In this regard, the cflow pointcut is a very good example of the kind of optimizations that can be achieved when scope constructs are part of the aspect language. Although cflow can be implemented as an automaton updated by a dedicated aspect (just like access control automata), it is translated, in most cases, into a simple counter which is increased at the beginning of join points matched by the argument of cflow, and

decreased at the end. Even more, a cflow pointcut can be removed completely when conjuncted with an always-false pointcut. This last optimization would be much more complicated without integrated support from the compiler.

Second, although one can devise optimizations to the access control automata supporting per-class restrictions, adding support for per-object restrictions—to take advantage of the increased expressiveness shown in Section 3.6.2—, makes optimizations much harder due to the dynamic nature of policy enforcement. The code to update the automaton for per-object restrictions is:

```
aspect FileRestriction{
  /* pointcut and advice remain the same */

  pointcut all(Object ctx): execution(* *.*(..)) && this(ctx);

  Object around(Object ctx): all(ctx){
    boolean t = isTrusted(ctx);
    if(t){ automaton.enterTrusted(); }
    else { automaton.enterUntrusted(); }
    try { return proceed(); }
    finally {
      if(t){ automaton.exitTrusted(); }
      else { automaton.exitUntrusted(); }
    }
  }}
```

When the transitions to take in the automaton depend on dynamic conditions, in this case whether the object currently in context is trusted or not, no simple general static analysis applies: every method in the system must be selected by the aspects to update the automata. Conversely, the access control scoping strategy is generic and *known in advance*. Actually, the conditions expressed by its components are all statically determinable, except for deciding whether a call to doPrivileged is a self call or not. Note that even this condition can be statically determined, by adopting a syntactic resolution of self calls, as in the encapsulation policies of Schärli *et al.* [66]. Therefore it is possible to evaluate the scoping strategy for access control at compile time, avoiding unnecessary runtime overhead.

3.7 Related Work

The relation between aspects and security has a long history. We described most of the related work in Section 2.4, which we do not repeat here. The only additional line of research missing from that description is alternative implementations of stack inspection, which we detail next.

In [84], Wallach *et al.* present SAFKASI, in which code is transformed to follow a *security-passing style* [85]. This is similar to continuation-passing style because an additional parameter representing the current access control state is added to all methods. The value of this parameter is maintained by a pushdown automaton. This is the work that inspired the access control automaton presented in Section 3.3.5 (as mentioned before, the only difference is that we use aspect orientation for updating the automaton). In contrast, our approach "passes" permission specifications around in the form of restriction aspects with an appropriate propagation strategy.

3.8 Summary

In this chapter, we have presented two completely modular access control architectures based on aspects, one using AspectJ, the *de facto* aspect-oriented extension to Java; and another using dynamic deployment and scoping strategies. To the best of our knowledge, this is the first successful attempt to fully modularize the access control architecture of Java by means of aspect orientation. Previous proposals only considered the modularization of access control checks, whereas we also deal with the more advanced features associated to the stack inspection algorithm, *i.e.* privileged execution and permission contexts.

We showed that, contrary to the current implementation of access control in Java and in our implementation using AspectJ, the scoping strategies approach is not based on specialized support for stack introspection, but rather uses general-purpose aspect-oriented constructs. The key element for this is the advanced control of aspect scoping provided by scoping strategies.

Availability. The elements used in this study are available at http://pleiad.cl/research/3sec. They include (1) the Scheme interpreter for a multi-threaded Scheme-like language with scoping strategies; (2) the TmpCleaner service implemented in this language; (3) the AspectJ implementation of the JAC architecture; and (4) the list of pointcuts for access control in the standard Java distribution.

Chapter 4

The Playground: AspectScript

As explained in the previous chapter, Java does not support scoping strategies. For this reason, we used a Scheme interpreter to experiment with ModAC. But we wanted to implement and test a practical library for access control, and at the same time, formally prove its security properties. The results of these efforts are described in Chapters 5 and 6 respectively. In this chapter[1], we present AspectScript, an aspect language for JavaScript, which supports scoping strategies and that served as a tool to achieve our goals.

4.1 Introduction

AspectScript builds upon advances in aspect-oriented language design research to address the specificities of JavaScript in a novel way. Logically, AspectScript integrates scoping strategies [72, 73] for proper control over the scope of dynamically-deployed aspects. But at the same time, AspectScript integrates other interesting features. Inspired by the work on aspects in higher-order procedural languages by Dutchyn, Tucker, and Krishnamurthi [26, 82], AspectScript supports first-class aspects; both pointcuts and advices are defined using first-class functions, providing the full benefits of higher-order programming. In line with this work and the inherently dynamic nature of JavaScript, AspectScript supports dynamic deployment of aspects, as found for instance in CaesarJ [7] and AspectScheme [26]. In addition, AspectScript avoids issues of infinite loops due to aspect reentrancy [71]. Finally, AspectScript not only supports implicitly-generated join points following the language model, but also provides the possibility to define custom join points triggered

[1] This chapter is based on the publication: "AspectScript: Expressive Aspects for the Web" [78].

explicitly, as in Ptolemy [62]. This combination of features is unique in the space of current aspect languages.

This chapter is organized as follows. Section 4.2 reviews the main elements of the AspectScript language in a systematic manner, and Section 4.3 presents key points regarding the implementation of the core of AspectScript. Section 4.4 discusses related work and Section 4.5 summarizes.

4.2 A Tour of AspectScript

We now describe AspectScript in more details, reviewing its hybrid join point model (Section 4.2.1), followed by the aspect model (Section 4.2.2), and the deployment model (Section 4.2.3). Finally, Section 4.2.4 describes how aspect reentrancy is controlled.

4.2.1 Hybrid Join Point Model

At its core, AspectScript adopts a join point model in the line of that of AspectJ [49], but tailored for the JavaScript language. As in any aspect-oriented language, join points are generated implicitly upon certain evaluation steps of the program. Pointcuts can then quantify over these join points. While quantification is a crucial feature of aspect languages, the obliviousness brought by implicitly-generated join points is more controversial. In particular, it suffers from a number of problems related to the difficulty of reconciling the (low) level of abstraction of standard join points with the need for quantifying over application-specific events. To this end, AspectScript also includes a mechanism for explicitly triggering custom join points, in the style of Ptolemy [62]. To the best of our knowledge, combining both implicit and explicit join point generation is a distinguishing feature of AspectScript in the current design space of aspect languages.

Standard join points. Figure 4.1 presents the standard join points supported in AspectScript. Save the last kind, which corresponds to custom events, all these join points are generated implicitly when a related expression is about to be evaluated. The JavaScript object model is peculiar in several respects. In particular, any function is considered a method of an object: top-level functions and anonymous functions are considered methods of the global root object of JavaScript. In addition, executing `this.foo()` in the context of an object `o`, where `foo` is a method of `o`, is different from invoking simply `foo()`. In the latter case, `foo` executes in the context of the global object, not of `o`. Also, a function is an object per-se, which can have properties on its own. AspectScript considers

join point	points in the program execution at which...
new	a function or object is created.
init	a function or object is initialized.
call	a function is called.
exec	a function is applied.
p-read	an object property is read.
p-write	an object property is written.
event	AS.event(..) is called.

Figure 4.1: Dynamic join points of AspectScript.

these peculiarities when generating join points, setting the target and current object properties of the join point appropriately.

Custom join points. In addition to standard, implicitly-generated join points, AspectScript also supports explicitly-generated custom join points. This mechanism corresponds to typed events in Ptolemy [62]. It addresses the limitation of implicit invocation in aspect-oriented languages, by making it possible to treat any (block of) expression(s) execution as an event that can be quantified over by pointcuts. Events have a type that closely corresponds to their intended (application-specific) semantics, rather than being tied to the base language operational semantics. It is also possible to communicate an arbitrary set of context information to other pointcuts and advices without introducing unnecessary coupling to program details. In AspectScript, typed events are supported as custom join point with a type attribute. This attribute can be a string (JavaScript does not support symbols) or any object. Because AspectScript does not modify the syntax of JavaScript, the (block of) expression(s) that corresponds to the custom join point is defined in a thunk. In addition to the type and the thunk, a custom join point has an arbitrary number of properties, which can then be used in poincuts and advices. Aspects perceive custom join points like standard ones. Similarly, if no aspect apply, the original computation takes place, *i.e.* the specified thunk is applied.

4.2.2 Higher-Order Aspects

AspectScript is directly inspired by AspectScheme [26], in which aspects, pointcuts, and advices are first-class values. Consequently, they can be created and manipulated at runtime. An aspect in AspectScript is a pointcut-advice pair; pointcuts and advices are plain JavaScript functions.

Pointcut model. Following standard practice, it is standard to define a pointcut as a function that takes a join point as parameter and returns an environment if it matches, or false if it does not [53].

```
function call(fun) { //call
  return function (jp,env) {
    return (jp.isCall() && jp.fun === fun)? env : false;
  } }

function exec(fun) { //execution
  return function (jp,env) {
    return (jp.isExec() && jp.fun === fun)? env : false;
  } }

function set(target,name) { //property write
  return function(jp,env) {
    return (jp.isPropWrite() && jp.target == target && jp.name == name)? env : false;
  } }

function cflow(pc) { //control flow
  return function cflow(jp,env) {
    return pc(jp,env)? env : cflow(jp.parent,env);
  } }

function not(pc) { //negation
  return function(jp,env) {
    return (pc(jp,env) != false)? true : false; //'not' ignores the env.
  } }
```

Figure 4.2: Some pointcuts available in AspectScript.

The environment is used by pointcuts to pass information to the corresponding advice. A peculiarity of AspectScript in this respect is that pointcuts are also parameterized by an environment. This permits inner pointcuts of a given pointcut to communicate using the environment. For instance, in the pointcut (pc1 && pc2), the environment returned by pc1 (if it matches) is then passed to pc2.

Figure 4.2 presents some of the pointcut designators available in AspectScript. Because pointcuts are standard JavaScript functions, their definition does not rely on any additional syntactic constructs. Also, pointcut designators take full advantage of the potential of higher-order functions. No standard pointcut in AspectScript modifies the environment. Pointcuts exposing typical contextual information like this, target, and args in AspectJ would be redundant here because these values are available as join points properties (*e.g.* jp.target, jp.args).

It is important to notice that AspectScript pointcuts do not rely on meta-data like types or annotations in order to match join points. The reason is that JavaScript does not support types nor annotations. AspectScript pointcuts also avoid using variable names to discriminate first-class values; rather they rely on value identity. This can be seen in the definitions of the call and exec pointcut designators in Figure 4.2, where the reference equality operator (===) is used to compare functions.

Deployment expression	Scope
deploy(asp) / undeploy(asp)	global
deploy([ss,] asp, fun)	on block: dynamic by default
deployOn([ss,] asp, val)	on object: lexical by default

Figure 4.3: Aspect deployment in AspectScript. Scoping strategies can be optionally specified in the last two alternatives.

Identity-based comparison is important in a language where functions are first-class values, and hence can be bound to many names, or none at all. Name-based selection in this context can result in many false negatives (*e.g.* a function execution not matched because it is applied through an alias) as well as false positives (*e.g.* a variable name that is used to refer to different functions at different moments in time).

Advice model. Advices in AspectScript are functions parameterized by a join point and an (optional) environment. The join point object passed as parameter has a proceed method, which permits the execution of the original computation at the join point. In the case of a custom join point, proceed evaluates the associated thunk. The environment corresponds to the environment returned by the associated pointcut. Like pointcuts, advices can access the bindings in the environment.

AspectScript supports a basic scheme for the composition of aspects: when several aspects apply over the same joint point, the applications of their advices are nested like in AspectJ. Like in AspectScheme, the precedence for the application of nested advices is determined by the order in which the aspects are deployed: the last-deployed aspect goes first. We come back on aspect weaving[1] in Section 4.3.2, when describing the implementation of AspectScript.

4.2.3 Deployment and Scoping Strategies

In this section we describe the AspectScript features associated to the deployment and scope control of aspects.

Dynamic Deployment. As in CaesarJ [7] and AspectScheme [26], AspectScript supports dynamic deployment of aspects. Deployment options are presented in Figure 4.3. The first option is global scope: deploy(a) deploys aspect a such that it sees all join points in the execution of the program, until it is explicitly undeployed with undeploy(a). Internally, deploy (resp. undeploy) just adds (resp. removes) an aspect to the global aspect environment, globalAspects.

[1]Recall from Chapter 1 that weaving is the process in which the aspects are inserted into the system at proper places.

In order to deploy an aspect with dynamic scope (*e.g.* fluid-around in AspectScheme), deploy can take a thunk (no-arg function) as an extra argument. In this case, the aspect sees all join points in the dynamic extent of the evaluation of the thunk. Internally, this variant of deploy temporarily adds the aspect to the global environment, executes the thunk, and then removes the aspect:

```
function deploy(aspect, fun){
  globalAspects.add(aspect);
  try{ return fun(); }
  finally{ remove(aspect); }
}
```

This mutation-based implementation is necessary because JavaScript does not support dynamic binding.

The third deployment option is per-value deployment, which follows the semantics of per-instance deployment in *e.g.* CaesarJ and AspectJ (per-this). The scope of an aspect deployed using deployOn is lexical, therefore the aspect only sees join points occurring lexically within method bodies of the objects it is deployed on. If deployed on a function, the aspect sees all join points in the body of the function, including inner functions.

Scoping Strategies. Both deployment on a block (deploy) and deployment on values (deployOn) can be refined using a scoping strategies, specified as an optional first parameter. A scoping strategy is a triple of functions [c,d,f]. c (resp. d) is propagation function specifying whether or not an aspect propagates along the call stack (resp. in newly-created functions or objects). f is an activation function making it possible to filter out certain join points. In other words, c permits to stop the unconditional propagation of dynamic scope at certain points, d enables aspects to be captured in certain procedural values as they are created, and f specifies a deployment-local refinement of the aspect pointcut.

All three functions are boolean-returning functions that take a join point as parameter[1]. In the case of c, the join point is the call the aspect can propagate through; in the case of d, the join point is the creation of the new object or function; and in the case of f, the join point is the one subject to filtering.

[1] In AspectScript, c, d and f can be specified directly as values; this is syntactic sugar for the corresponding constant function.

4.2.4 Control of Aspect Reentrancy

Thus far we have ignored a fundamental issue with the proposed design of AspectScript: if a function application triggers a join point, and a pointcut is a plain JavaScript function, then applying a pointcut pc triggers a function application join point against which pc should itself be evaluated, leading to an infinite loop! And the same happens with an advice that applies a function whose application is matched by its associated pointcut. In fact, while this issue of *aspect reentrancy* is exacerbated in a higher-order aspect language like AspectScript, it is latent in any aspect language[1]. Some mechanism must be provided to avoid aspects potentially matching join points triggered by their *own* execution [13, 71].

Limitations of current solutions. Higher-order procedural aspect-oriented languages like AspectML and AspectScheme adopt different solutions to this problem, though both rely on a mechanism to deactivate weaving in some way. Indeed, AspectScheme uses a primitive operator app/prim to apply a function without generating a join point, and AspectML suggests a similar disable primitive that hides the whole computation of an expression. The difference is their scope: app/prim only hides a single function application join point, but does not hide the computation triggered by that application; conversely, disable has dynamic scope. Current AspectJ patterns to address these issues are similar to disable: adding a control-flow condition to pointcuts such that join points occurring in the dynamic extent of advice execution are ruled out [13]. The AspectJ pattern however does not work in the case of reentrancy caused by if pointcuts [71].

As argued extensively elsewhere [74], relying on control flow checks to avoid reentrancy is flawed for several reasons. Most importantly, since proceed is called as part of an advice, reasoning on control flow conflates advice computation and base computation, resulting in aspects not applying when then should. Also, disabling weaving for aspectual computation simply makes it impossible for aspects to advise aspects.

Solution in AspectScript. In order to properly address issues of reentrancy without entailing conflation or sacrificing visibility of aspect computation, AspectScript relies on the notion of *execution levels* [74], a refinement of the proposal of stratified aspects [13]. In a nutshell, the idea is to structure computation into levels, starting with base computation at level 0. Aspect computation is by default considered as ocurring at level 1, and is therefore invisible to other aspects. If needed, aspects can be deployed at higher levels of execution, thereby possibly observing other aspects, or an aspect can explicitly lower part of its computation (if so, a mechanism ensures that it does not see

[1] http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html

its *own* computation). A detailed description of the implementation of reentrancy control and execution levels is however outside the scope of this work. In-depth motivation and formal description of execution levels can be found in [74].

The bottom line is that, for the programmer, AspectScript just works as if the issue of reentrancy did not exist, *precisely* because AspectScript handles execution levels behind the scene.

4.3 Implementation

In this section we detail the implementation of AspectScript. In particular, Section 4.3.1 details the code transformation phase required for weaving, and Section 4.3.2 describes the weaving process. Finally, we end with a preliminary analysis of AspectScript performance in Section 4.3.3.

Given the dynamic nature of JavaScript, weaving in AspectScript is mostly done at runtime. In order to be able to dynamically weave aspects without modifying a particular JavaScript engine, our AspectScript implementation first performs a code transformation phase in which some expressions are rewritten into invocations of *reifiers* (Section 4.3.1). These reifiers then make it possible to perform runtime aspect weaving (Section 4.3.2). Relying on code transformation means that AspectScript can run on any client browser, without requiring the installation of dedicated complements or add-ons. The complete transformation phase is implemented in JavaScript, using an optimized version of the parser of the Narcissus project[1]. A JavaScript implementation is interesting in order to be able to do client-side parsing; this could be useful for example in the case of remote code loading or `eval` invocations. These applications are subject of further exploration.

4.3.1 Code Transformation

The code transformation phase rewrites JavaScript source code[2] by introducing invocations of *reifiers*. Reifiers are functions that generate the join points associated with the rewritten expression.

For the reader unfamiliar with JavaScript, Figure 4.4 presents a subset of its syntax, relevant to the transformation performed by AspectScript. A script is a list of statements, which can be either a function or variable declaration, a block or an expression. Expressions include object creation

[1] http://mxr.mozilla.org/mozilla/source/js/narcissus/

[2] AspectScript per se does not extend the *syntax* of JavaScript; it is provided as a library.

$$
\begin{array}{rrcl}
\mathit{Script} & z & ::= & \overline{s} \\
\mathit{Statement} & s & ::= & \mathsf{function}\ id(\overline{id})\ \{\overline{s}\} \mid \mathsf{var}\ id = e \mid \{\overline{s}\} \mid e; \\
\mathit{Expression} & e & ::= & \mathsf{new}\ e_c(\overline{e}) \mid \{\overline{id : e}\} \mid [\overline{e}] \mid \mathsf{function}(\overline{id})\{\overline{s}\} \\
 & & & \mid e_f(\overline{e}) \mid e_t.id(\overline{e}) \\
 & & & \mid id \mid e.id \mid id = e \mid e_t.id = e
\end{array}
$$

Figure 4.4: Subset of the JavaScript syntax.

(either with a constructor or literally), definitions of arrays and anonymous functions, function invocation and variable/property access and assignment.

The transformation is defined by the syntax-driven rewriting function $[\![\cdot]\!]$, presented in Figure 4.5. The first four rules deal with statements. Rule 1 recursively triggers the transformation of the expression being used to initialize the declared variable. A function declaration is rewritten into a var declaration, binding the function name to the (transformation of) the anonymous function definition (rule 2). Rule 3 groups functions declared in a block, and moves them to the beginning of that block. This reordering is necessary because of rule 2: in JavaScript a variable needs to be *explicitly assigned* to a value in order to be used, whereas a function declaration makes the function name available globally in its declaring block. Finally, rule 4 rewrites the expression part of the statement.

The next rules rewrite expressions. The general scheme is the same: all the information required to generate a join point is gathered and passed as parameters to the appropriate reifier, which then triggers weaving (Section 4.3.2). Object creation (rule 5) is rewritten into an invocation of the ρ_{new} reifier, passing as argument a first-class anonymous function that encapsulates the actual instantiation (parametrized by the constructor and actual arguments). This function is then used at runtime for the proceed method of the constructed join point. The same approach is used for array creation (rule 6). The reifier also receives the constructor and the arguments. In order to support generation of function execution join points, functions are wrapped; thus, rule 7 uses a different reifier ρ_{wrap}. Finally, literal object creation (rule 8) uses yet another reifier, ρ_{lit}. This reifier receives as argument a first-class anonymous function that encapsulates the initialization of the properties specified in a given literal object creation.

The remaining rules are straightforward. Function invocations (rules 9 and 10) are transformed into invocations of the ρ_{call} reifier, passing as parameters the target of the call, the function being invoked, and its arguments. Rule 9 uses the global object as a target because the JavaScript semantics specifies that when no target is specified when invoking a function, the global object must be used.

Statements

$$[\![\text{var } id = e]\!] = \text{var } id = [\![e]\!] \quad (1)$$
$$[\![\text{function } id_f(\overline{id})\ \{\overline{s}\}]\!] = \text{var } id_f = [\![\text{function } (\overline{id})\ \{\overline{s}\}]\!] \quad (2)$$
$$[\![\{\overline{s}\}]\!] = \{[\![\overline{s}_{function-decls}]\!][\![\overline{s}_{other-decls}]\!]\} \quad (3)$$
$$[\![e;]\!] = [\![e]\!]; \quad (4)$$

Object Creation

$$[\![\text{new } e_c(\overline{e})]\!] = \rho_{new}(\text{function}(k,\overline{a})\{\text{return new } k(\overline{a})\}, [\![e_c]\!], [[\![\overline{e}]\!]]) \quad (5)$$
$$[\![[\overline{e}]]\!] = \rho_{new}(\text{function}(k,\overline{a})\{\text{return } [\overline{a}]\}, \text{Array}, [[\![\overline{e}]\!]]) \quad (6)$$
$$[\![\text{function}(\overline{id})\ \{\overline{s}\}]\!] = \rho_{wrap}(\text{function}()\{\text{return function}(\overline{id})[\![\{\overline{s}\}]\!]) \quad (7)$$
$$[\![\{\overline{id : e}\}]\!] = \rho_{lit}(\text{function}()\{\overline{\text{this}.id = [\![e]\!]}\}) \quad (8)$$

functionction Invocation

$$[\![e_f(\overline{e})]\!] = \rho_{call}(\text{AS.globalObject}, [\![e_f]\!], [[\![\overline{e}]\!]]) \quad (9)$$
$$[\![e_t.id(\overline{e})]\!] = \rho_{call}([\![e_t]\!], [\![e_t.id]\!], [[\![\overline{e}]\!]]) \quad (10)$$

Property Access

$$[\![e.id]\!] = \rho_{read_p}([\![e]\!], \text{"}id\text{"}) \quad (13)$$
$$[\![e_t.id = e]\!] = \rho_{assign_p}([\![e_t]\!], \text{"}id\text{"}, [\![e]\!]) \quad (14)$$

Figure 4.5: Rewriting function.

Rules for property access (13 and 14) transform reads and writes of properties into invocations of the ρ_{read_p} and ρ_{write_p} reifiers, passing as parameter the name of the property being accessed, and in the case of a property write, the value being assigned. The owner of the property is passed as the first argument to both reifiers.

4.3.2 Runtime Weaving

At runtime, the calls to the reifiers inserted by rewriting are evaluated. Apart from creating the corresponding join point, the reifiers also trigger the weaving process by invoking the weave function of AspectScript. To illustrate reifiers, below is the (simplified) implementation of the ρ_{call} reifier:

```
function r_call(obj, fun, args){
  var jp = new CallJP(obj, fun, args, currentJoinPoint);
```

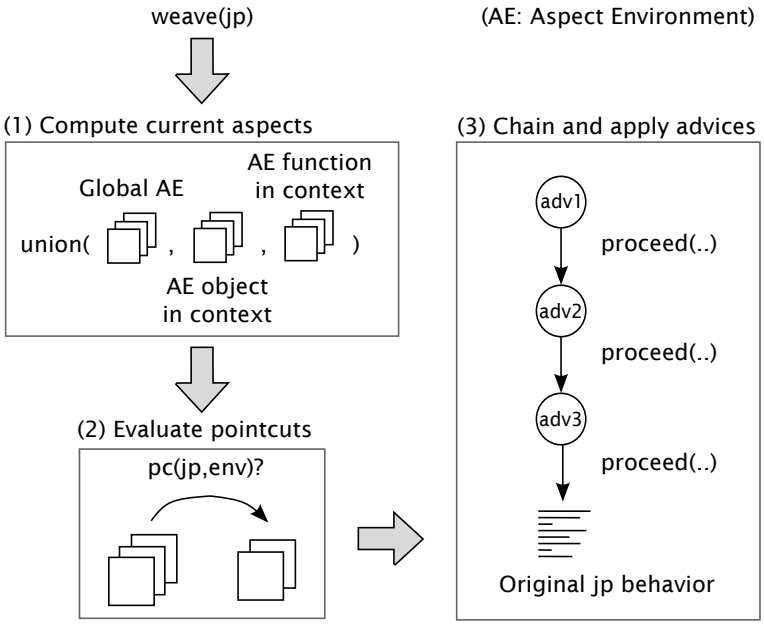

Figure 4.6: Weaving process.

```
  return weave(jp);
}
```

The r_call function receives as arguments the target of the call (obj), the function being invoked (fun), and the arguments to that function (args). This is the JavaScript code executed to weave a call expression, corresponding to the transformation rules 9 and 10 in Figure 4.5.

The weave function weaves the join point it receives as argument. Figure 4.6 depicts the weaving process initiated by weave, and its simplified definition is as follows:

```
function weave(jp){
  // 1. compute current aspects
  var currentAspects = union(aspectsIn(ctxObj),aspectsIn(ctxFun),globalAspects);
  // 2. evaluate pointcuts
  var advices = match(jp, currentAspects);
  // 3. chain and apply advices
  return chainAndApply(advices);
}
```

The first step is to determine the set of aspects that may potentially apply. Recall from Section 4.2.3 that AspectScript supports different deployment mechanisms: global, per-function, and per-object. In consequence, at a given join point, the list of aspects that may apply is the *union* of the aspects that are deployed in several aspect environments (step 1). First, aspects may have been

deployed with deployOn, therefore each object and function has its own aspect environment. At a given join point, aspects in the currently-executing function (ctxFun) and the currently-execution object (ctxObj) have to be considered (accessed with aspectsIn in the code above). Then, the *global* aspect environment contains all aspects deployed using deploy (either deployed globally or with dynamic scope). This global aspect environment also contains the aspects propagating through the stack due to scoping strategies semantics (it is potentially modified at the entrance and exit of every call join point).

Once the list of current aspects has been determined, pointcuts of all these aspects are evaluated against the current join point (step 2). The advices of aspects that matched the current join point are then chained together, each one nesting the next one (step 3). Executing proceed in one advice triggers the following advice; in the last advice in the chain, proceed runs the original base computation. Note that following AspectScheme [26], before and after advices are only syntactic sugar for around advices. Aspects are deployed in the reverse order of their deployment, like in AspectScheme.

4.3.3 Runtime Performance

The primary design goal of AspectScript is expressiveness. When conceiving the language, we have not sacrificed any potentially valuable feature on the basis of its expected cost. Still, we are interested in making AspectScript a practical solution for aspect-oriented programming in JavaScript; it therefore makes sense to evaluate its overhead. In order to do so, we selected a subset of the jQuery [46] test suite (+1300 tests, around 90% of the total number of tests) that includes only CPU-intensive tests, as this constitutes a worst-case scenario for AspectScript. For this experiment, we used jQuery 1.3.2 (118KB of source code) on an Intel Core 2 Duo, 2.66 GHz PC with 2GB of RAM running Ubuntu 9.04 (kernel 2.6.28) and Firefox 3.5.2. Running the selected tests takes approximately 20s in this setting, without AspectScript.

We then transformed the jQuery library as described in Section 4.3.1 in order to support aspects. The transformed code ended up weighing 326KB (a 2.8x factor). Just running the test suite with the transformed library, without any aspect weaving at all, is 5.7 times slower. This overhead includes the generation of approximately 3.4 million join points (Mjp). Considering this baseline, Figure 4.7 describes the overhead introduced by an aspect following different deployment scenarios: *(a)* a globally-deployed aspect, *(b)* an aspect deployed with the pervasive scoping strategy (*i.e.*[true,true], recall Section 3.6.2.3), and *(c)* an aspect deployed on an object (the $ object, entry point of the

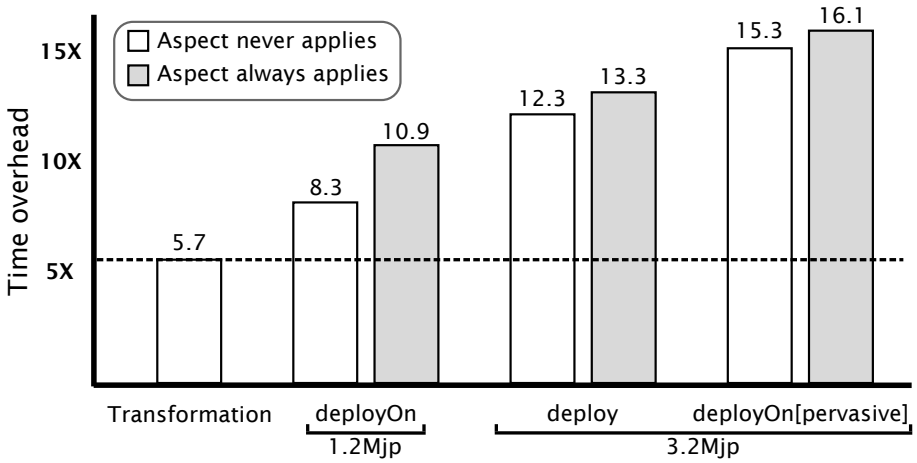

Figure 4.7: Performance overhead of AspectScript for CPU-intensive tests in the JQuery test suite.

jQuery library). In both cases *(a)* and *(b)*, the deployed aspect sees all 3.4Mjp; in case *(c)*, due to lexical scoping, the aspect only sees 1.2Mjp. The advice of the aspect only calls proceed on the join point. For each case, we measured the overhead with both a pointcut that never matches and a pointcut that always matches, thus giving a lower and an upper bound for the overhead of the presence of the aspect. We did experiments where the pointcut does a function comparison (using reference equality) instead of just returning a boolean, without any noticeable difference.

The overhead of a global aspect is between 12.3x (never match) and 13.3x (always match). Activating scoping strategies and deploying a pervasive aspect implies a relative overhead of 20% only (15.3x-16.1x). Using a per-object aspect (which results in the aspect seeing only 1.2Mjp) reduces the overhead down to 8.3x-10.9x. The results are overall encouraging, especially considering the low amount of time we spent working on optimizing the implementation so far; our focus has rather been to get the semantics right and working. We believe there are many venues for optimization to be explored, both in the runtime itself and by adding the possibility for the programmer to statically declare certain aspects and restrictions on the general dynamicity of AspectScript. Finally, raw overhead in a CPU-intensive scenario does not really reflect the fact that JavaScript is more widely used for interactive applications, that may even include remote communication. For instance, we have tested AspectScript (global aspect always matching) on a JavaScript Tetris game, without any noticeable difference[1].

[1]Both versions of the Tetris game can be tested online on the AspectScript website.

4.4 Related Work

Modularization of crosscutting concerns has long been considered in Web technologies [68]. An example of this is the separation of an HTML document in different sources, such as CSS files for presentation style and JavaScript files for functionality. However, within these sources, crosscutting concerns are still present. For this reason, diverse tools to modularize these concerns are available in the Web. For instance, the jQuery library [46] allows programmers to separate crosscutting concerns in the DOM of a Web page, *e.g.* for adding borders to all tables. We now review AOP frameworks for JavaScript, as well as AOP proposals for other higher-order procedural languages.

AOP for JavaScript. A large number of lightweight AOP frameworks for JavaScript have been made available on the Web by programmers, that rely on function wrappers. In the simplest case, there is no quantification at all, and one explicitly has to give both a function and the advice function that should be added to it with wrapping [61]. Some frameworks make it possible to wrap methods of an object by specifying the names of the methods to wrap [8, 9, 43], and others support regular expressions for describing method names to wrap [3, 19].

AOJS [87] is a more mature AOP framework that takes quantification more seriously, and relies on code transformation. However, it does not embrace the features of JavaScript as AspectScript does. Aspects (with before and after advice only) are specified in a separate XML file; therefore, aspects cannot enjoy the full power of higher-order programming. In addition, function identification is name-based, rather than identity-based. As discussed in Section 4.2.2, this leads to fragile pointcuts, that may (not) match when expected, and excludes the execution of anonymous functions. The join point model of AspectScript is also considerably richer, and extensible. AOJS does not support dynamic deployment of aspects. While this is beneficial in terms of performance, it is limitating in terms of expressiveness. We plan to improve performance of AspectScript in the future by supporting static specifications to restrict full dynamicity to a certain extent. Finally, aspect reentrancy in AOJS is avoided by simply deactivating weaving during pointcut and advice evaluation, an insufficient solution (Section 4.2.4).

AOP for higher-order procedural languages. AspectScript draws significantly on previous work on AOP extensions of higher-order procedural languages like Scheme, Standard ML and Caml[1].

AspectScheme [26] is an aspect-oriented extension of Scheme, which is dynamically typed, like JavaScript. In order to support the full power of higher-order programming, pointcuts and advices

[1]We do not repeat here the limitations of the mechanisms provided by AspectScheme and AspectML to avoid reentrancy (Sect. 4.2.4).

in AspectScheme are standard functions, aspects are dynamically deployed (either with dynamic or lexical scope), and function identification is identity-based rather than name-based. AspectScript imitates AspectScheme in all these dimensions. This said, AspectScript supports a richer join point model, including custom join points. Context exposure in AspectScript is more powerful, allowing pointcuts in a composition to use bindings exposed by prior pointcuts. Finally, AspectScript supports more expressive aspect deployment (deployOn) and scoping (scoping strategies).

AspectML [22] is an AO extension of Standard ML [55]. Like AspectScript, pointcuts are first-class values, but advices are not. Aspects are deployed with lexical scope only. AspectML does not use function identity to match pointcuts; instead, pointcuts use function names and argument types to match functions that are currently in lexical scope. Anonymous functions cannot be advised (although an as-yet-unsupported any keyword is discussed). As discussed before, relying on function names in a language where functions are first-class values easily leads to both false positives and false negatives.

Aspectual Caml [54] extends Caml [51] with aspect-oriented constructs. As well as AspectML, Aspectual Caml integrates well with the type system of the host language. In addition, Aspectual Caml takes into account the fact that Caml supports object-oriented programming. In essence, the aspect-oriented features of Aspectual Caml are very similar to AspectML, except for the fact that Aspectual Caml supports pattern matching over method names (in any scope), as well as advising anonymous functions. However pointcuts are not first-class values.

4.5 Summary

Because JavaScript is being widely used in applications of ever-increasing complexity, it is particularly relevant to propose a powerful aspect-oriented extension. AspectScript is a first concrete, working step in this direction. AspectScript fully embraces the characteristic features of JavaScript, by supporting higher-order aspects, a full-fledged join point model, customizable quantified events, and dynamic aspect deployment with expressive scoping. Aspect reentrancy is avoided without causing any burden on programmers. This combination of features is unique in the current design space of aspect languages.

In the following chapters we make use of AspectScript to experiment with access control in JavaScript: we first implement ModAC in JavaScript (Chapter 5) and then analyze its security properties (Chapter 6).

Availability. The elements mentioned in this chapter are available at http://www.pleiad.cl/aspectscript/. They include (1) the AspectScript library; and (2) the Tetris game mentioned in Section 4.3.3.

Chapter 5

ZAC: ModAC for Javascript

Until now, we have presented ModAC just as a design, and the only concrete realization of this design is a Scheme interpreter (Chapter 3). In order to implement and test a real ModAC implementation, we developed AspectScript, an aspect-oriented extension to JavaScript supporting scoping strategies (Chapter 4). In this chapter[1] we present ZAC, our practical library for access control in JavaScript. ZAC is an easy-to-use implementation of ModAC in AspectScript.

5.1 Introduction

The ModAC approach to modular access control presented in Chapter 3 is only a design. The only artifact related to it is a proof-of-concept implementation based on a Scheme interpreter for a multi-threaded Scheme-like language with scoping strategies. This chapter extends our work on ModAC with two elements:

- We show the design and implementation of ModAC/AS, the instantiation of ModAC in JavaScript, and ZAC, a practical library for JavaScript based upon ModAC/AS (Section 5.2)
- We describe ZAC (standing for "easy access control"), a practical library for JavaScript based on ModAC/AS, featuring an easy-to-use API, hiding its aspect-oriented foundations from the developer (Section 5.3)

[1] This chapter is based on two publications: "Secure and Modular Access Control with Aspects" [81] (first part); and "Access control in JavaScript" [80] (second part).

We present related work in Section 5.4 and a summary in Section 5.5.

5.2 ModAC/AS Design & Implementaion

In this section we briefly summarize the details of the ModAC/AS implementation, focusing on restriction aspects (Section 5.2.1), the access control scoping strategy (Section 5.2.2), and the bootstrapping process (Section 5.2.3).

5.2.1 Restriction Aspects

As explained before, a restriction aspect works by adhering to the following pattern: the pointcut selects accesses to a sensitive resource (just like a permission aspect), but the advice immediately aborts the access by not proceeding with the primitive operation; scoping strategies are used to ensure that the aspect only *sees* forbidden accesses. Restriction aspects in ModAC/AS are plain AspectScript aspects, for instance:

```
var netRestriction = {
  pointcut : function(jp){ return jp.kind == NEW && jp.fun === XMLHttpRequest; },
  advice : function(jp){ throw "Cannot access the net."; }
};
```

This aspect forbids the access to the network. Its pointcut identifies instantiations of XMLHttpRequest objects, and the advice throws an exception with an informative message. Another possibility is not to throw an exception but to silently abort the sensitive resource access. For instance:

```
var alertRestriction = {
  pointcut : function(jp){ return jp.kind == EXEC && jp.fun === alert; },
  advice : function(jp){ /* do nothing */ }
};
```

This restriction aspect simply skips the execution of the alert method, in order to avoid popups. The key is to not proceed with the original computation associated to the join point.

5.2.2 The Scoping Strategy for Access Control

As explained in Chapter 3, restriction aspects are limited to see only illegal resource accesses by means of scope control. The scoping strategy for access control in ModAC/AS that supports basic permission checking, privileged execution, and permission contexts is:

```
var acs = [ //access control strategy
  function(jp){ return !(jp.fun === doPrivileged && jp.target === jp.context);},
  function(jp){ return jp.target instanceof ACContext; }
];
```

The call stack propagation function expresses both basic permission checking and privileged execution. Essentially, it specifies that a restriction aspect always propagates on the call stack, except on privileged calls. A privileged call is a *self call* to doPrivileged. This way, a restriction aspect propagating through the stack stops its propagation upon a privileged call, and hence does not see resource accesses that occur in the control flow of that call. Considering self calls for privileged execution permits to maintain the aspects of the object initiating the action.

The delayed evaluation propagation function expresses the capture of permission contexts. It ensures that restriction aspects propagate to instances whose prototype is ACContext; therefore, creating such an object is a means to take a snapshot of the restriction aspects present at that point in time. Later on, it is enough to include these objects in the stack to restore the permission context. This is done by an overloaded version of doPrivileged that accepts an ACContext as extra parameter—as described in Chapter 3.

5.2.3 Bootstrapping Access Control

Since access control is fully modularized, it is just one more aspect. In order for it to be effective in a given system, it has to be activated. In a language with dynamic aspect deployment, the only way is to do so explicitly in the program (*e.g.* around the main method, around the loading of a script, etc.). In a language with static deployment, access control must still be equivalently activated (*e.g.* command line or configuration file).

In the case of ModAC/AS, the activation of access control is performed by wrapping the main program in a deployment of the ACDeployer aspect (Figure 5.1). ACDeployer ensures that the relevant parts of the activity of all objects are under control of restriction aspects. It does so by deploying

```
1  var ACDeployer = {
2    acs: ..., //access control strategy
3    pointcut: function(jp){ return jp.kind == NEW; }, // creation of objects
4    advice: function(jp){
5      var obj = jp.proceed();
6      var restrictions = getRestrictionsFor(obj);
7      deployOn(acs, restrictions, obj); //per−object deployment
8      return obj;
9    } };
10 deployOn([false,true],ACDeployer, function(){ /* main program */ });
```

Figure 5.1: Deployer aspect for deploying restriction aspects.

these restriction aspects on newly-created objects with the access control scoping strategy acs defined previously. Crucially, the deployment of restriction aspects must be done exactly in between the creation of an object and the beginning of its initialization. This way, when the object initiates computation, the necessary restriction aspects are already deployed on it.

The ACDeployer aspect deploys restriction aspects on objects when they are created. First, its pointcut matches all object creations (line 3). Then, the advice (lines 4-8) deploys the corresponding restriction aspects on the newly-created object (line 5), using deployOn (line 7) and specifying the access control scoping strategy (line 2). Finally, the object is returned (line 8). The set of restriction aspects that corresponds to a particular object is determined by the getRestrictionsFor method (line 6). This method abstracts the process of determining the needed restrictions. A possible implementation is to mimic the access control architecture of the JVM by returning the restriction aspects that correspond to the permissions declared in a policy file. Another implementation is to return restrictions based on dynamic conditions, such as the kind of user currently interacting with the application, as in role-based access control [32]. Line 10 deploys ACDeployer such that it propagates in all created objects (delayed evaluation is set to true); this ensures that it sees all object creations.

5.3 ZAC: Practical Access Control for JavaScript

ZAC is our practical library for access control in JavaScript. ZAC is implemented according to what is described in Section 5.2, but featuring a simpler API, hiding the complexity associated to aspects from the developer (access to the underlying power of aspect-orientation is still available for advanced developers, as described in Section 5.3.6). ZAC is based on assigning access control

policies to scripts when loaded in the browser. At runtime, the policy is enforced for every action performed by these scripts.

The principle of ZAC is that foreign code can use every feature of JavaScript, including eval, and also can access every reference to any object in the system. However, the access control policy assigned to the foreign code when loaded forbids dangerous actions before they happen at runtime.

5.3.1 Loading Scripts

Enforcing access control policies with ZAC is very easy through the use of a simple API. For example, to load a third-party script it is only necessary to use the ZAC.load method:

```
ZAC.load("http://www.evilsite.net/evil.js", ZAC.newDefaultPolicy());
```

Loading the evil.js script using load, its execution is automatically subject to the restrictions in the policy specified as the second argument. Policies in ZAC are sets of restrictions aspects (just *restrictions* hereafter). Figure 5.2 shows the restrictions in the policy returned by ZAC.newDefaultPolicy(), targeted to restrict the access to common sensitive resources. For instance, the ZAC.R_ALERT restriction forbids the use of alert dialogs. These dialogs are normally used to provide valuable information to the user, but they can also be used to turn a page (or even the whole browser) unusable by endlessly showing an alert dialog. Another example is ZAC.R_LOCATION, that forbids redirections of the page. This restriction prevents malicious scripts from sending the browser to potentially dangerous sites.

5.3.2 Policy Enforcement

When loaded using ZAC.load, a script is unable to bypass the specified access control policy, directly or indirectly. This means that the script itself will not be able to perform any action forbidden by the policy, and also that it will not be able to lead other (possibly trusted) code to do it on its behalf. As explained in Chapter 3, this is due to the deployment of restriction aspects with the access control scoping strategy.

Figure 5.3 shows four different attempts to call the alert function, all ending with an exception raised by the access control policy. The first one is a direct call. The second one uses delegation: the untrusted code invokes the (trusted) info function, which in turn tries to call alert. These two

Constant (ZAC.)	Description
R_ALERT	prevents alert calls.
R_LOCATION	prevents redirection of the browser.
R_C_STYLES	prevents calls to computedStyles().
R_INNER_HTML	sanitizes strings assigned to the innerHTML property.
R_COOKIES	prevents access to cookies.
R_GLOBAL	prevents access to properties of the global object.
R_EVAL	prevents arbitrary use of eval (only JSON deserialization).
R_FUN	prevents instantiations of Function objects.
R_STO_SI	prevents calls to setTimeout and setInterval with a string argument.
R_HTTP_REQ	prevents instantiations of XMLHttpRequest objects.
R_DEF_PROTOS	prevents modification of prototypes of default objects.
R_ARGS	prevents access to the arguments property of other functions.
R_WATCH	prevents calls to watch and unwatch.
R_UNENCR	prevents calls to toSource and uneval.
R_ZAC_POLICIES	prevents access to ZAC policies.

Figure 5.2: Restrictions in ZAC's default access control policy. Constants are accessed as properties of the ZAC global property.

attempts end with an exception because the restriction is propagated when the evil.js script invokes alert. The third attempt is interesting, because it uses eval. ZAC ensures that restrictions of the code that calls eval are inherited by the eval-ed code: therefore, this attempt also fails. Finally, the last attempt is more intricate: although ZAC's policies are specified at the level of scripts (loading them using load), they are enforced at the level of individual objects. The consequence is that if an object is created during the execution of a script subject to a certain policy, that object's execution will always be subject to that policy: wherever the object goes, the policy follows it. Therefore, in the example, the policy is present in the stack when the anonymous function calls info. This is the reason why the third attempt also ends with an exception.

5.3.3 Defining Custom Policies

As typical sets, policies in ZAC can be modified by adding or removing restrictions (using the add and remove methods respectively). There are several scenarios where this can be useful. For instance, it seems natural to specify different policies for different foreign scripts, depending on the level of confidence the host page has in each of them. Another scenario is when a policy must be constructed programmatically (*e.g.* according to the preferences of the user). The API of ZAC

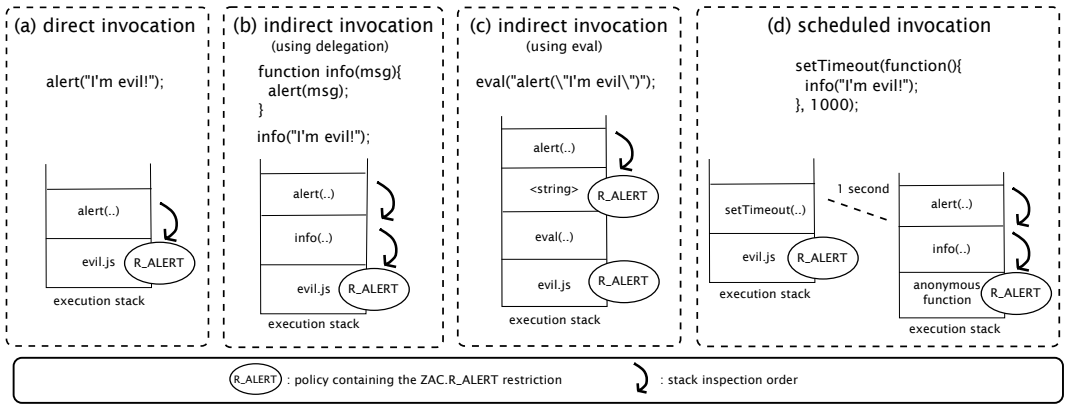

Figure 5.3: Policy enforcement in ZAC. All these attempts will fail with an exception raised by the access control policy.

permits programmers to modify policies even *after* using them to load a script. These modifications will affect all the scripts already loaded using the policy.

In addition to default policies, empty policies can be created using ZAC.newPolicy(). In the following piece of code, two policies are configured to enforce different restrictions in two different scripts:

```
var softPolicy = ZAC.newPolicy(); //fresh policy (no restrictions)
//restrict only page redirections and evaluation of code in notSoEvil.js
softPolicy.add(ZAC.R_LOCATION, ZAC.R_EVAL);
ZAC.load("http://evilsite.net/notSoEvil.js", softPolicy);

//get a default policy and remove only the R_ALERT restriction for evil.js
var hardPolicy = ZAC.newDefaultPolicy();
hardPolicy.remove(ZAC.R_ALERT);
ZAC.load("http://evilsite.net/evil.js", hardPolicy);
```

5.3.4 Extending ZAC

Each restriction in a policy is targeted to restrict the access to common sensitive resources. ZAC's policies can be extended by adding new restrictions targeted to protect other resources.

Event Name	Properties	Common properties	Common methods
New	fun, args	parent	proceed(args), clone(), is<eventName>()
Init	target, fun, args		
Call	target, fun, args, context, reflective		
Exec	target, fun, args		
PropRead	target, name		
PropWrite	target, name, value		

Figure 5.4: Properties and methods of events supported by ZAC. `fun`: the function being used as a constructor (`New` and `Init`), or being called/executed (`Call` and `Exec`). `args`: the arguments of the event. `target`: the target of the event. `context`: the object performing the call. `reflective`: whether the call was performed using `call` or `apply`. `parent`: the parent event (like a stack of execution). `proceed`: executes the event. `clone`: clones the event (useful to store a reference). `is<eventName>()`: boolean-returning utility methods to identify the kind of event.

5.3.4.1 Defining Restrictions

A restriction is a JavaScript object with two properties, both of which are functions. A rule property is in charge of identifying the access to the resource, and an action property is in charge of specifying the action to take when the resource access occurs[1]. For example, the ZAC.R_ALERT restriction is implemented as follows:

```
ZAC.R_ALERT = {
  rule : function(event){ return event.isCall() && event.fun === alert; },
  action: function(event){ throw "Cannot call alert"; }
};
```

The function bound to the rule property identifies calls to the alert function by returning true when such calls occur. It uses the event parameter, which is a representation of the event occurring in the script. Figure 5.4 shows the complete list of event types supported by ZAC, and the corresponding properties (fields and methods) in each case. The function bound to the action property simply throws an exception because invoking alert is completely forbidden. Notice that the function bound to the rule property of a restriction is evaluated only when an object subject to that restriction is/was involved in the execution, and not for every event in the application.

An alternative to simply throwing an exception is to provide, possibly under certain circumstances, an alternate "safe" behavior. For example, the eval function of JavaScript is widely considered dangerous because it permits to execute arbitrary, potentially malicious, code. However,

[1] We use rule and action instead of pointcut and advice because rule/action are closer to the security domain.

eval has a very useful application: the deserialization of a JSON [47] object from a string. Because JSON object's serialization format do not permit functions, no arbitrary code can be executed when evaluating a serialized object. In other words, using eval for deserializing JSON objects is safe. The code below shows the ZAC.R_EVAL restriction that only forbids the evaluation of code that is not in JSON format:

```
ZAC.R_EVAL = {
  rule: function(event){ return event.isCall() && event.fun === window.eval; },
  action: function(event){
    try{
      return JSON.parse(event.args[0]);
    }
    catch(e){
      throw "Eval can only be used to deserialize JSON objects.";
    }
  } };
```

Just like in the ZAC.R_ALERT restriction, the rule property identifies calls to a certain function, in this case, eval. The action property, instead of immediately throwing an exception, first tries to evaluate the first argument of eval (event.args[0]) as a JSON string. If it effectively is a JSON string, the resulting object is returned. Otherwise, an exception is thrown by JSON.parse; the restriction action then throws an exception itself.

5.3.4.2 Adding New Restrictions

ZAC comes with a set of predefined restrictions (recall Figure 5.2), which corresponds to common cases. It is possible to define whole new kinds of restrictions as well. Let us define a new restriction that limits the number of windows a script can open:

```
function nWindowsRestriction(n){
  return {
    nWindows: 0,
    rule: function(event){ return event.isCall() && event.fun === document.open; },
    action: function(event){
      if(++this.nWindows > n){
        throw "Cannot open more than " + n + " windows.";
      }
      return event.proceed();
    }
  }; }
//add the restriction to a fresh policy
```

```
var policy = ZAC.newPolicy().add(nWindowsRestriction(3));
```

The code above shows the use of three interesting elements: a stateful restriction, a restriction factory, and the use of event.proceed().

Stateful restrictions. A restriction can have any number of additional properties apart from rule and action. In the example, the nWindows property is used to keep track of the number of windows opened by the script. Therefore the restriction returned by nWindowsRestriction is called a *stateful* restriction. Adding a stateful restriction to more than one policy implies that its state is shared among these policies. For this reason, a restriction like the one returned by nWindowsRestriction will allow three windows in total, suming up all the windows opened by all scripts the restriction applies to.

Restriction factory.. To define a restriction, creating an object with the appropriate properties suffices. However, using a restriction factory like the nWindowsRestriction function has two advantages. First, it permits to easily parameterize the restriction (the n argument in the example). And second, it permits to obtain a different instance each time the generator is invoked. This can be used to avoid the sharing issue of stateful restrictions.

The proceed method. The proceed method can be used to execute the original behavior the event parameter represents in the script. In the example, the *call* to the open function. The proceed method accepts the same parameters as the original event. If specified, these parameters replace the original parameters passed to the event. If omitted, the event is executed with the original parameters.

5.3.5 Privileged Execution

ZAC supports the notion of privileged executions. A privileged action is started by a self call to doPrivileged, and the code to execute in this privileged context is specified as the body of the function passed as parameter:

```
this.doPrivileged(function(){
  // privileged code
});
```

The semantics of ZAC for privileged execution defines that any self call to doPrivileged starts a privileged action. Non self calls to doPrivileged are not considered privileged executions. In consequence, an untrusted object cannot call doPrivileged on a trusted object to bypass access control. Of

course, because ZAC identifies privileged actions by the name doPrivileged, programmers must be careful not to use this method name inadvertently.

5.3.6 Taking Advantage of AspectScript

As mentioned before, AspectScript's aspects corresponds exactly to ZAC's restrictions, where the rule property is the pointcut, and the action property is the advice. ZAC depends only on AspectScript to be secure: if AspectScript generates all the events associated to the execution of all untrusted objects, these objects will not be able to do anything without ZAC being aware of it. This is the reason why ZAC policies cannot be circumvented.

The fact that ZAC is implemented on top of AspectScript brings many benefits, of which we highlight two: the ample variety pointcuts included in AspectScript, and the ability to reason about the program execution.

AspectScript includes several predefined pointcuts that can be composed to identify more intricate actions. For example, the following pointcut identifies the calls to alert that occur inside the body of any method of obj:

```
var PCs = AspectScript.Pointcuts;
var pc = PCs.call(alert).and(PCs.within(obj));
```

Pointcuts in AspectScript can also match sequences of events, optionally restricted by temporal conditions. This can be useful in access control to, for instance, forbid the invocations to alert that occur too frequently, say more than one per second.

An example of a security property based on actual computation—which is actually checked by most browsers—is to ensure that a third-party script does not degrade the interactive experience of the user: if a script takes too long to execute, the browser suspends its execution and asks the user whether to simply abort the script. While such a restriction is beyond the realm of capabilities, it is straightforward to define with ZAC:

```
function nInstructionsRestriction(n){
  return {
    nInstructions: 0,
    rule: function(event){
      return ++nInstructions > n;
    },
    action: function(event){
```

```
        if(confirm("This script is running for too long. Abort it?")){
          throw "Cannot execute more than " + n + " instructions";
        }
        nInstructions = 0;
        return event.proceed();
      }
    }; }
```

For simplicity, we use instruction count rather than actual time. The point here is just to give an idea of the wide range of policies that can be expressed using dynamic AOP as a foundation for defining restrictions.

5.3.7 Performance Overhead

To give an idea of the performance overhead introduced by the enforcement ZAC policies, we selected the CPU-intensive tests of the jQuery library [46], and applied different policies to them. These tests generate more than 4.4 million of events (property accesses, method calls, object creations, etc.) in approximately 20 seconds when executed without any restriction. Applying all the restrictions of Figure 5.2 makes the tests execute 15.5 times slower. Applying only the R_ALERT restriction reduces the overhead factor to just 3.8.

In order to test ZAC in real-world scenarios, we have developed a small prototype Firefox plugin based on it. This plugin mimics the NoScript plugin (noscript.net), which can forbid the execution of potentially malicious code. Our extension, with proper control of the generated events, is usable on a large number of websites.

5.4 Related Work

The only way provided by the HTML standard to control the rights of external code is to use frames: including a script inside a frame gives full permissions to the script, but only over its enclosing frame. However, in the majority of cases, this option is not expressive enough, mainly because it is too coarse-grained. For instance, it is impossible to grant certain permissions to external code, like showing alert dialogs, while forbidding others, like accessing cookies.

For this reason, several proposals have been presented for controlling the actions that scripts can

Name	Enforcement	Based on	Extensible/ Specification	Granularity
AdSafe	static	static analysis	no/-	script
FBJS	static + dynamic	object wrappers	no/-	page
BrowserShield	dynamic	program monitoring	yes/blacklisting	page
Caja	static + dynamic	object capabilities	yes/whitelisting	script
Delimited Histories	dynamic	program monitoring	yes/blacklisting	origin
ZAC	dynamic	dynamic AOP	yes/blacklisting	object

Figure 5.5: JavaScript proposals for access control.

perform within a web page (Figure 5.5). AdSafe [2] limits the JavaScript features external code can use, leaving only a "secure" subset of the language that can be statically checked to ensure it does not perform potentially dangerous actions. FBJS [29] rewrites the code to replace references to standard objects with limited, but equivalent objects, preventing any potentially dangerous action. This is what Facebook uses for third-party applications. BrowserShield [63] is a more flexible proposal that can be used in more scenarios. BrowserShield transforms code to make it trigger notifications of its own activity during execution. An observer entity then decides whether the activity of the code should be allowed or not. New policies can be specified as functions using the JavaScript language itself. Caja [17] is a more principled approach to access control based on the object-capability model. In this model, external code can access (and therefore use) references to other objects only if the host page provides them. If no reference is provided, the external code can still compute based on its own (harmless) references. Caja is used in iGoogle and My Yahoo!. Finally, in [64], objects are assigned an *ownership record* according to their origin. When objects with different ownership records interact, their action are recorded and called *delimited history* (delimited because not all history is recorded). Then, in the so called *suspension points* (critical points, such as a call to eval), the runtime system asks the access control policy whether the history of actions should be allowed. If not, the runtime system reverts all the effects of these actions. Policies are specified in C++ and dynamically linked to the browser engine to achieve good performance and to prevent attacks to the access control architecture.

As depicted in Figure 5.5, ZAC combines very interesting features:

- **Dynamic enforcement of policies** enables the execution of programs that are safe but use unsafe constructs. For example, programs that use eval only to deserialize JSON objects from

strings, and not to execute arbitrary code, are safe. However, because they use a potentially unsafe construct, such programs are rejected by systems like AdSafe.

- Being based on **dynamic aspect-oriented programming (AOP)** [27], allows ZAC policies to reason about program execution in its entirety. For instance, it is possible to define a policy that prevents a script from never ending, or taking too much time for acceptable web interaction. This kind of policy is impossible to express in the object-capability model because the property does not depend on object references, but rather on computation itself. Delimited histories makes it possible to reason about the past, but only a subset of it.

- **Extensible access control specifications** is crucial considering that different usage scenarios imply different requirements. This is acknowledged by other proposals like BrowserShield, Caja and delimited histories.

- **Object-level granularity** is a unique feature of ZAC. Allowing the coexistence of different policies for different scripts within a Web page is fundamental. Going beyond the script/origin level down to the object level also enables a secure interaction among scripts: objects from one script can use objects from other scripts, possibly with different policies; the correct policy will be unequivocally enforced; even objects from the same script can be further restricted under certain conditions (if they are passed as parameters, compromising its properties for instance). This is not the case with capabilities, where an untrusted object that obtains a reference to a sensitive resource can use it without limitations. This kind of security issue is equivalent to forget enforcing a particular restriction, in this case, R_ALERT.

5.5 Summary

In this chapter, we have presented the design and implementation of ModAC/AS, and ZAC, a practical library for JavaScript based upon ModAC/AS. This implementation was materialized in AspectScript, our aspect-oriented extension to JavaScript, described in the previous chapter.

We compared ZAC to existing solutions for access control in JavaScript. We concluded that apart from key features such as dynamic enforcement of policies and extensible policies, ZAC is the only approach supporting object-level permissions, a critical feature for more expressive access control.

Availability. ZAC is available at http://pleiad.cl/aspectscript/zac.

Chapter 6

Securing ModAC

We presented ModAC in Chapter 3, instantiating this approach in Chapter 5 using AspectScript, the language we designed and implemented in Chapter 4. However, until now we have answered only part of our research question: access control can be aspectized and the evidence is ModAC. In this chapter[1] we tackle the two remaining issues. First, the formulation of ModAC is informal; its actual effectiveness in controlling accesses to sensitive resources has not been proven. Second, it leaves open the possibility for untrusted aspects to interfere with access control aspects, thereby ruining its effectiveness.

6.1 Introduction

None of the solutions presented until now are immune to security threats: not the approaches presented in Section 2.4 nor ModAC itself. The reason is that they implicitly assume that no other entities can affect the behavior of access control aspects. This chapter presents how we extend ModAC with three elements:

- We show that it is possible to fully modularize access control as an aspect, even in the presence of untrusted aspects, thanks to a *self-protecting* restriction aspect that impedes untrusted aspects to interfere with critical access control components (Section 6.3).
- We develop λ_{AS}, a core calculus for AspectScript based on λ_{JS} [40] for modeling JavaScript, and a variation of the semantics of LAScheme [74] for aspect weaving (Section 6.4).

[1] This chapter is based on the publication: "Secure and Modular Access Control with Aspects" [81].

- We state and prove the effectiveness and non-interference properties of an instantiation of ModAC in λ_{AS}, ModAC/λ_{AS}. The formulation of these results is detailed in Sect. 6.5. We discuss the extension of the results to ModAC/AS and other aspect languages in Section 6.6.

Section 6.2 introduces a classification of the possible threats to ModAC. Section 6.7 describes related work and Section 6.8 concludes.

6.2 Threats to Modular Access Control

ModAC seems to be a proof by existence that access control *can* be fully modularized using aspects, provided the aspect language supports a sufficiently expressive scoping mechanism. However, our previous work does not provide any formal guarantee in this respect. Most importantly, it does not consider threats posed by the presence of other, possibly untrusted, aspects.

Attacker Model. We consider a simple attack model where the attacker can define an aspect whose purpose is to defeat access control. For this purpose, the malicious aspect may use any of the following techniques:

- Interfere with the evaluation of the pointcuts of access control aspects
- Interfere with the evaluation of the advices of access control aspects
- Manipulate the state (data structures, properties, etc.) access control aspects rely on

However, the attacker cannot:

- Alter the specifications of access control aspects
- Alter the specification of the access control policy (*i.e.* alter which entities are considered trusted or untrusted)
- Alter the runtime environment, including the aspect weaver

The actual way of materializing these attacks depends greatly on the base language. For instance, in AspectJ it is impossible to match a pointcut execution whereas in JavaScript, a pointcut is simply a function.

Inhibition. Following the same attack model, De Borger *et al.* showed how easy it is to interfere

with access control by means of aspects [23]. For instance, this AspectJ aspect completely inhibits access control in Java programs:

```
public aspect MaliciousAspect{
  void around(): execution(void SecurityManager+.check*(..)){ }
}
```

As opposed to the JVM and the CLR, ModAC does not exhibit the previous vulnerability, simply because there are no explicit calls to a stack inspection algorithm. However, there are other alternatives for untrusted aspects to inhibit access control, to which ModAC is vulnerable: *i.e.* to prevent access control components—restriction aspects, the access control strategy, and the ACDeployer aspect (see Section 5.2)—from actually controlling accesses.

We introduce the distinction between implicit and explicit inhibition. *Implicit inhibition* is based on using the aspect weaving mechanism to inhibit access control, such as in the above AspectJ example. *Explicit inhibition* consists of using other means provided by the base language (*e.g.* side effects) to prevent the different components of the access control system to fulfill their role.

Explicit inhibition. There are many kinds of explicit inhibition, depending on the considered programming language. In a purely functional language, it is impossible to alter a function or mutate existing bindings and data structures. But in a stateful world, risks exist if the state of the access control components can be aliased and mutated. Such risks are exacerbated in languages like JavaScript, where it is possible to dynamically remove object members.

Fortunately, explicit inhibition requires the malicious entity to perform explicit actions, which can be observed and prevented by dedicated restriction aspects. For instance, the following restriction forbids any action on the netRestriction restriction we defined in Section 5.2.1 (*e.g.* modification of its properties, invocation of its methods):

```
var metaNetRestriction = {
  pointcut: function(jp){ return jp.target === netRestriction; },
  advice: function(jp){ throw "Cannot manipulate the netRestriction aspect"; }
};
```

For any kind of explicit inhibition, a dedicated restriction must be defined. This shows how ModAC elegantly protects itself from explicit inhibition.

Implicit inhibition. Because explicit inhibition can be prevented by means of restriction aspects, this chapter focuses on implicit inhibition. Indeed, implicit inhibition is peculiar because it is di-

rectly enabled by the use of an aspect-oriented language; also, implicit inhibition can be achieved in any aspect language, regardless of whether or not the language allows arbitrary effects.

In the case of ModAC, there are three kinds of implicit inhibition: pointcut inhibition, advice inhibition, and scoping strategy inhibition. Pointcut inhibition consists in preventing the pointcut of an access control component from matching at relevant join points. For instance, the following malicious aspect inhibits the pointcut pc of a restriction aspect:

```
var maliciousAspect = {
  pointcut: function(jp){ return jp.kind == PCEXEC && jp.fun === pc; },
  advice : function(jp){ return false; }
};
```

The other kinds of inhibition follow a similar pattern: making a pointcut return false as above, making an advice do nothing by matching its execution but never proceeding, or impeding propagation of restriction aspects by making their propagation functions return always false, etc.

6.3 R̊: One Aspect to Rule them All

In this section we present R̊ (pronounced "ring"), a self-protecting restriction aspect that prevents untrusted aspects from inhibiting access control in ModAC. We first introduce some terminology to discriminate different kinds of aspects (Section 6.3.1). We then describe and justify our design goals for secure modular access control (Section 6.3.2), and a general approach to control untrusted aspects (Section 6.3.3). We finally present R̊ and explain how it prevents inhibition of both access control and itself (Section 6.3.4).

6.3.1 Aspect classification

First, we refer to all aspects that are part of ModAC—restriction aspects as well as the ACDeployer aspect—as *access control aspects*. We then make the distinction between *trusted aspects*, which should be given unrestricted freedom; and *untrusted aspects*, which are potentially trying to inhibit acces control. Classifying aspects as trusted or untrusted depends on the access control policy of a given application. For example, a possible policy consists in considering all aspects defined in local

code as trusted, whereas aspects defined in remote code are deemed untrusted. We do not commit to any specific means to express this classification.

In addition, we introduce a set of *protected aspects*. By definition, this set contains all aspects whose inhibition must be prevented. In order to secure ModAC, this set must include all access control aspects (but is not restricted to those aspects).

6.3.2 Securing ModAC: design goals

Our design goals for secure and modular access control are as follows:

G1 *The base language must be completely oblivious to access control.*

G2 *Untrusted aspects must not inhibit protected aspects, but are otherwise free to advise any join points.*

G3 *Trusted aspects should be able to advise any join point.*

The first goal (G1) is the *raison d'être* of ModAC. Beyond being an important validation for AOP itself, fully modularizing access control with aspects allows access control to be added to other aspect languages, without requiring ad hoc support for it. The other two design goals are concerned with securing ModAC without overly restricting the programming model.

Design goal (G2) states that the non-inhibition property must be achieved without simply ruling out untrusted aspects. Untrusted aspects must be able to do whatever their access policies specify; the only strong requirement is that they do not inhibit protected aspects.

Design goal (G3) states that trusted aspects should be able to see any join point. This goal discards a restrictive approach that prohibits any kind of weaving (trusted or not) in certain core classes—thereby strongly coupling access control and weaving.

For instance, the Aspect-Oriented Permission System (AOPS) [23] ensures non-inhibition by disallowing any kind of weaving at join points lexically located in access control aspects and other sensitive components such as permission classes and the PermissionManager class. Doing so impedes even trusted aspects to advise these classes. In addition, it means that the weaver (and hence the aspect language semantics) is specifically tailored to take access control into account, something that we discard as of design goal (G1). Therefore, AOPS violates two of our design goals,

(G1) and (G3).

6.3.3 Preventive inhibition

In order to reconcile goals (G2) and (G3)—*i.e.* preventing the inhibition of protected aspects by untrusted aspects, while allowing trusted aspects to see any join point—we introduce a simple technique: *preventive inhibition*. Preventive inhibition consists in inhibiting untrusted aspects *before* they get a chance to inhibit protected aspects.

To achieve preventive inhibition, it is sufficient to ensure that untrusted aspects do not apply at join points occurring in the control flow of protected aspects. For restriction aspects, this means that untrusted aspects cannot interfere with the identification of resource accesses nor with the process of aborting these accesses. For the ACDeployer aspect, this means that untrusted aspects cannot interfere with the identification of object creations nor with the calculation and deployment of restriction aspects.

How can preventive inhibition be achieved while maintaining (G1), *i.e.* without requiring modifications to the aspect language semantics? A tentative answer is to slightly update untrusted aspects by conjuncting their pointcuts with the following:

```
!cflow(function(jp){ return protectedAspects.contains(jp.target); })
```

This effectively makes the pointcuts of untrusted aspects evaluate to false for join points that are in the control flow of join points whose target is in the set of protected aspects.

It is hard to reconcile this global transformation of all untrusted aspects with (G1). Indeed, the required global transformation can be performed by means of general-purpose constructs, such as the global pointcut proposed by the abc team [10]. However, identifying untrusted aspects may depend on knowledge only available at runtime, especially in dynamic languages like JavaScript.

6.3.4 The R̊ restriction

How can preventive inhibition be achieved while maintaining (G1), *i.e.* without requiring modifications to the aspect language semantics? We describe a simple solution that is essentially just a programming pattern of ModAC. The approach relies on using a specific restriction aspect, called R̊. R̊ is in charge of preventive inhibition for all protected aspects, including itself, thereby fulfilling

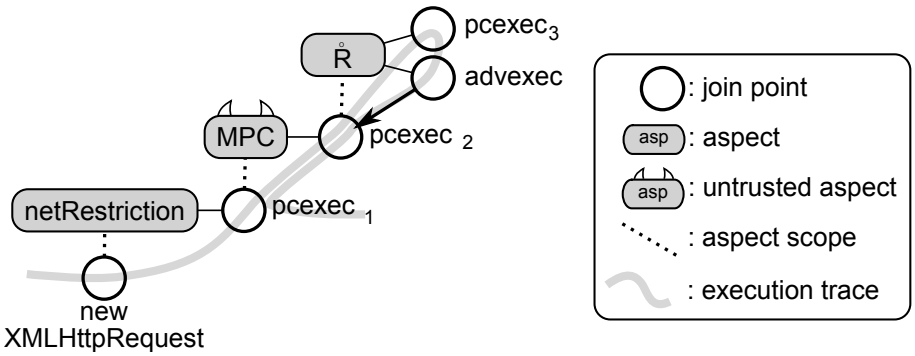

Figure 6.1: Pointcut inhibition prevented by R̊.

goals (G2) and (G3). Because R̊ is a restriction like any other, goal (G1) is fulfilled as well: there is no need to change the language semantics to support it.

Inhibition with R̊. R̊ is deployed on untrusted objects at creation time, just like other restriction aspects. Its definition is:

```
var R̊ = {
  pointcut: function(jp){
    return jp.kind == PCEXEC &&
          cflow(function(jp){ return protectedAspects.contains(jp.target); });
  },
  advice: function(jp){ return false; }
};
```

R̊ inhibits every pointcut execution it sees, provided that the execution is in the control flow of a join point whose target is in the protected aspects set. In consequence, all aspects in the protected aspects set cannot be inhibited by untrusted aspects, simply because untrusted aspects do not even get a chance to see the join points they would potentially advise. Note that R̊ is the first-class equivalent of the pointcut conjunction discussed in the previous section. Making it a restriction aspect like any other is the key to enforce this inhibition check without affecting the language semantics.

Illustration. Figure 6.1 illustrates how R̊ avoids pointcut inhibition by an untrusted aspect MPC on the netRestriction aspect presented before. When a new XMLHttpRequest instance is created, a join point is generated. The netRestriction aspect sees this creation, and therefore, its pointcut is evaluated. This generates a pointcut execution join point ($pcexec_1$), which is observed by MPC. Consequently, the MPC pointcut is evaluated, which generates another pointcut execution join point ($pcexec_2$). Since MPC is untrusted, R̊ was deployed on it. Hence, the pointcut of R̊ sees $pcexec_2$, and

matches it (it is a pointcut execution join point and a protected aspect, netRestriction, is in the control flow). In consequence, R̊ inhibits the pointcut of MPC. Advice and scoping strategies inhibitions are prevented in a similar way.

Self-protection. Crucially, R̊ can protect *itself* from inhibition by untrusted aspects, following the exact same principle. To do so, R̊ is added to the set of protected aspects. Self-protection of R̊ can be observed in the same Figure 6.1, by replacing the reference to netRestriction on the figure with R̊. An untrusted aspect can try to inhibit R̊ as many times as it wants in the same flow of execution. If the interaction is infinite, the program does not terminate[1]. If the interaction is finite, R̊ eventually rules the untrusted aspect. Self-protection of R̊ elegantly secures ModAC by not introducing any additional mechanism; R̊ is just a restriction aspect protecting access control aspects, including itself, and other protected aspects, from inhibition by untrusted aspects.

Bootstrapping. R̊ uses the protectedAspects set to identify the aspects it must protect from implicit inhibition. Naturally, untrusted entities must not be allowed to interfere with this data structure. Inhibiting access control by interfering with the protectedAspects set can either be done implicitly via weaving, or through explicit manipulation. Implicit inhibition is already prevented by R̊ itself (because the protectedAspects is manipulated only by entities pertaining to the set itself, like R̊). Explicit inhibition is avoided by using a dedicated restriction aspect:

```
var paRestriction = {
  pointcut: function(jp){ return jp.target === protectedAspects; }, //any action on protectedAspects
  advice : function(jp){ throw "Cannot manipulate the protected aspects set"; }
};
```

This restriction follows the same pattern as the metaNetRestriction presented before; it forbids *any* action over protectedAspects. This restriction must be deployed on all untrusted entities at creation time. Note that this restriction is just another restriction, and therefore (G1) is still fulfilled.

6.4 A Calculus for AspectScript

The previous section has informally explained how ModAC can be made secure thanks to the R̊ restriction aspect. However, ModAC itself has never been proven to be effective, even in the absence of untrusted aspects; and it remains to be proven that R̊ is effectively securing ModAC.

[1] Any untrusted piece of code is (a priori) given the power of the base language (which is Turing-complete) and can therefore always provoke non-termination. Different mechanisms can be used to avoid this misbehavior (*e.g.* timeout, limit on the number of produced join points), we proposed a solution (with restriction aspects!) in Section 5.3.6

$$
\begin{array}{rrcl}
\mathit{Value} & v & ::= & c \mid \textbf{fun}(x\cdots)\{e\} \mid o \mid l \\
\mathit{Bool} & b & ::= & \textbf{true} \mid \textbf{false} \\
\mathit{Const} & c & ::= & n \mid \mathit{str} \mid b \mid \textbf{undefined} \mid \textbf{null} \\
\mathit{Object} & o & ::= & \{\mathit{str} : v \cdots\} \\
\mathit{Expr} & e & ::= & x \mid v \mid \textbf{let}\,(x = e)\,e \mid e(e\cdots) \mid e[e] \mid \\
 & & & e[e] = e \mid e = e \mid \textbf{ref}\,e \mid \textbf{deref}\,e \\
\mathit{Store} & \mu & ::= & \epsilon \mid \mu + (l \mapsto o)
\end{array}
$$

$n \in \mathscr{N}$, the set of numbers; $\mathit{str} \in \mathscr{S}$, the set of strings;
$x \in \mathscr{X}$, the set of variable names; $l \in \mathscr{L}$, the set of locations.

Figure 6.2: Syntax of the λ_{JS} language (excerpt; slightly modified).

In order to do so, we focus on AspectScript and establish a formal basis for it: the λ_{AS} calculus. Section 6.5 then states formally that ModAC/λ_{AS}, the implementation of ModAC in the λ_{AS} calculus, is correct and secure. Note that Section 6.5 is also accessible to readers who prefer not to dive into the formal semantics, as it includes a precise description of the argumentation line.

λ_{AS} is a core calculus for AspectScript, and as such has to be faithful to JavaScript. We use the λ_{JS} calculus [40] as a starting point. The aspect-oriented part of AspectScript is based on the calculus of LAScheme [74], which models first-class aspects with dynamic deployment and execution levels in a higher-order procedural language. ModAC however also requires (a form of) scoping strategies, which are not part of the existing LAScheme formalization. As a result, even though we simplify the treatment of execution levels in the calculus, λ_{AS} is more complex. But this complexity is directly drawn from the required characteristics of the language (JavaScript-based, first-class aspects with dynamic deployment, execution levels, and scoping strategies).

We now first give a brief overview of λ_{JS}, and then describe its extension to support aspect weaving with dynamic aspect deployment and scoping strategies.

6.4.1 Core JavaScript: λ_{JS}

Guha *et al.*designed λ_{JS} as a core subset of JavaScript to which JavaScript programs are desugared. The interest of λ_{JS} is its compactness. We briefly describe the syntax of λ_{JS}, the desugaring process, and a few reduction rules.

Syntax. Figure 6.2 shows part of the syntax of λ_{JS}. The language has primitive values such as numbers, strings, booleans, and two special values **null** and **undefined**, in addition to functions

(**fun**) and objects o. Objects are a series of attribute-value pairs enclosed in curly braces. Expressions include identifiers, values, a **let** construct, function application, property access, and property write. In order to support first-class mutable references, values are augmented with store locations. Objects in the store are explicitly referenced and dereferenced using **ref** and **deref**, respectively. λ_{JS} also includes typical control operators and primitive n-ary operators; we omit these for brevity.

Desugaring. Several JavaScript constructs are specified via translation (called "desugaring") to λ_{JS} [40]. For example, the desugaring of function creation is:

```
desugar⟦ function(x···){e} ⟧ = ref {
      "code": fun(this, fthis, x···) { desugar⟦ e ⟧ },
      "prototype": ref {"__proto__": (deref Object)["prototype"]}}
```

A function is desugared into an object (using the {...} notation) with two attributes: code and prototype. The code attribute is the actual function (note that function is a JavaScript term, and **fun** is a λ_{JS} term). Also, this is an ordinary identifier: it is the first formal parameter of a desugared function. In JavaScript, a method is a function, which is a value, and can be shared between objects; this refers to the currently-executing object. For the sake of properly dealing with aspect environments in λ_{AS}, we slightly extend λ_{JS} and pass a second parameter to every desugared function; the parameter, named fthis, is bound to the function object thus created by the desugaring process. Note that desugaring reveals some of JavaScript peculiarities: the prototype attribute of a function object is an object whose prototype is the prototype attribute of Object.

The semantics of λ_{JS} is defined as a small-step reduction relation $\hookrightarrow$. A program configuration $\langle \mu, e \rangle$ consists of a store and an expression. The reduction relation is standard. Evaluation contexts [92] are used to specify a call-by-value, left-to-right evaluation semantics. E.g., the reduction rule for object creation is:

$$\langle \mu, E[\textbf{ref}\ \{str : v \cdots\}] \rangle \hookrightarrow \langle \mu', E[l] \rangle \qquad \textsc{New}$$
$$\text{where } l \notin dom(\mu) \text{ and } \mu' = \mu + (l \mapsto \{str : v \cdots\})$$

ref simply allocates a new location in the store and returns it.

The function application rule is the standard β_v reduction:

$$\langle \mu, E[\textbf{fun}(x \cdots)\{e\}(v \cdots)] \rangle \hookrightarrow \langle \mu, E[e[v \cdots / x \cdots]] \rangle \qquad \textsc{Call}$$

$$
\begin{array}{rrcl}
 & J & ::= & \epsilon \mid j + J \\
 & j & ::= & \lceil k, l_o, l_f, p \rceil \\
 & k & ::= & \texttt{new} \mid \texttt{call} \mid \texttt{exec} \mid \texttt{pc-exec} \mid \texttt{adv-exec} \\
\end{array}
$$

$p \in \mathscr{T}$, the set of thunks $J \in \mathscr{J}$, the set of join point stacks

$$
\begin{array}{rrcl}
\mathit{Expr} & e & ::= & \ldots \mid \texttt{jp}\,(j, \alpha) \mid \texttt{in-jp}\,(e) \mid \texttt{c/asp}\; k\; e\; e \cdots \\
\mathit{EvalCtx} & E & ::= & \ldots \mid \texttt{in-jp}\,(E) \mid \texttt{c/asp}\; k\; v \cdots\; E\; e \cdots \\
 & v & ::= & \ldots \mid \boxed{J}
\end{array}
$$

Figure 6.3: Join points

6.4.2 AspectScript Semantics

We now describe the syntax and operational semantics of λ_{AS}, a core calculus for AspectScript based on λ_{JS}. Its operational semantics is defined via the reduction relation $\hookrightarrow$: $\mathscr{M} \times \mathscr{A} \times \mathscr{J} \times \mathscr{E} \rightarrow \mathscr{M} \times \mathscr{A} \times \mathscr{J} \times \mathscr{E}$.

We extend the λ_{JS} configuration with two additional elements: a λ_{AS} program configuration $\langle \mu, \alpha, J, e \rangle$ consists of a store $\mu \in \mathscr{M}$, an aspect environment $\alpha \in \mathscr{A}$, a join point stack $J \in \mathscr{J}$, and an expression $e \in \mathscr{E}$. The *stack aspect environment* α is used to maintain the aspects propagated through the stack by means of the call stack propagation function.[1]

In the following we describe the semantics of join points, aspects and their deployment, as well as the weaving semantics. The formalism is based on the semantics of LAScheme [74], an aspect-oriented Scheme-like language with execution levels, itself based on a combination of Clifton and Leavens's work [20] (modeling of the join point stack) and Dutchyn *et al.* [26] (weaving semantics). By convention, when we introduce new user-visible syntax (*e.g.* the aspect deployment expression), we use **bold** font. Internal terms are written in `typewriter` font.

6.4.2.1 Join Points

The join point stack J is a list of *join point abstractions* j, which are tuples $\lceil k, l_o, l_f, p \rceil$ (Figure 6.3). We introduce five kinds of join points: `new` for object creation, `call` for function application and method invocation, and `exec`, `pc-exec`, `adv-exec` for function, pointcut, and advice execution, respectively. Figure 6.4 describes the different values for the components of join point abstractions,

[1] We also maintain the currently-executing object/function in the program configuration, omitted here for simplicity. The online Redex model includes the full configuration.

k	`new`	`call` / `exec` / `pc-exec` / `adv-exec`
l_o	object prototype	target object
l_f	**null**	target function
p	primitive operation	

Figure 6.4: Join point abstraction attributes per kind.

depending on their kind. For instance, p is always the primitive operation (used to perform the original computation); l_o denotes the prototype of the object being created in a `new` join point, and the target object for `call` and the three execution join points.

In order to keep track of the join point stack in the semantics we introduce two internal expression forms. `jp` (j, α) introduces a join point j whose underlying computation via proceed will be executed with aspect environment α. `in-jp` (e) keeps track of the fact that execution of e is proceeding under a dynamic join point. We extend the definition of evaluation contexts accordingly (Figure 6.3). The expression `c/asp` (which stands for "call/aspect") is used later to treat pointcut and advice execution join points similarly. It is a function application annotated with the kind of join point k that needs to be created; this expression form is generated by the weaver, discussed later on.

A join point abstraction captures the minimum context information necessary for ModAC to work (target object and function), as well as to trigger its corresponding computation when necessary (the p function). We write $\boxed{J}$ to denote the reification of the join point stack J as a λ_{AS} value. A number of introspection primitives are provided; for instance, **kind** ($\boxed{J}$) is the λ_{AS} equivalent of jp.kind in AspectScript. Similarly, **tobj** (resp. **tfun**) can be used to retrieve the (location of the) target object (resp. function).

6.4.2.2 Aspects and Deployment

For the sake of conciseness and simplicity, we make the three following simplifications to λ_{JS} in this work: *i)* scoping strategies have constant boolean components (instead of join point predicates); *ii)* only per-object deployment (**deployOn**) is described; *iii)* we do not account for context exposure (*i.e.* pointcuts simply return **true** if they match, instead of an environment). These simplifications do not affect the validity of our results: constant propagation functions are enough to state and prove the desired properties of ModAC, **deployOn** is strictly more expressive than **deploy** [76], and context exposure is an orthogonal feature for this work.

$$
\begin{array}{rrcl}
Expr & e & ::= & \ldots \mid \textbf{deployOn}[e,e](e,e) \\
EvalCtx & E & ::= & \ldots \mid \textbf{deployOn}[E,e](e,e) \mid \textbf{deployOn}[b,E](e,e) \mid \\
 & & & \textbf{deployOn}[b,b](E,e) \mid \textbf{deployOn}[b,b](v,E)
\end{array}
$$

$$
\begin{array}{rrcl}
AspectEnv & \alpha & ::= & \alpha + (b_c, b_d, l) \mid \epsilon \\
Store & \mu & ::= & \epsilon \mid \mu + (l \mapsto o^{\alpha}) \\
 & \texttt{asps}(l) & = & \alpha, \text{ where } \mu(l) = o^{\alpha}
\end{array}
$$

$$
\begin{array}{l}
\langle \mu, \alpha, J, E[\textbf{deployOn}[b_c, b_d](l_{asp}, l_{obj})] \rangle \quad \textsc{DeployOn} \\
\hookrightarrow \langle \mu', \alpha, J, E[l_{obj}] \rangle \\
\text{where} \quad \mu(l_{obj}) = o^{\alpha'} \text{ and } \mu' = \mu(l_{obj} \mapsto o^{\alpha' + (b_c, b_d, l_{asp})})
\end{array}
$$

Figure 6.5: Aspects and deployment.

As described on Figure 6.5, an aspect environment α is a list of tuples (b_c, b_d, l) where l denotes the reference to the aspect, and the two boolean values corresponds to the `c` and `d` components of the scoping strategy specified at deployment time. An aspect can be any object whose pointcut attribute is a function that takes a join point stack as input and produces either **true** or **false**. To compensate for the absence of context exposure from pointcuts, an advice function also receives as first argument the current join point stack. An advice proceeds using the **proceed** ($\boxed{J}$) primitive.

An aspect is deployed with **deployOn**. Because **deployOn** embeds an aspect within an object, the stack aspect environment of the program configuration is not enough; each object needs to have its own aspect environment as well. To do so, we annotate an object o with its aspect environment α as o^{α}. By construction, an object is annotated with its aspect environment as soon as it is allocated in the store (with **ref**). We therefore extend the definition of the store, and introduce an internal function `asps` in order to access the aspects of an object in the store (see Figure 6.5).

The DEPLOYON rule shows the semantics of per-object deployment: the aspect (at location) l_{asp} is added at the end of the aspect environment of the object (at location) l_{obj}, along with the specified scoping strategy components.

$\langle \mu, \alpha, J, E[\textbf{ref}\ \{str : v \cdots\}] \rangle$ NEW
$\hookrightarrow \langle \mu, \alpha, J, E[\texttt{jp}(\lceil \texttt{new}, proto, \textbf{null}, p \rceil, \alpha)] \rangle$
where
$proto = v_i$ if $str_i =$ "__proto__"
$\alpha' = (\texttt{asps}(\texttt{cobj}()) \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha)|_d$
$p = \textbf{fun}()\{\ \texttt{new/prim}\ \{str : v \cdots\}^{\alpha'}\ \}$

$\langle \mu, \alpha, J, E[\textbf{fun}(x \cdots)\{e\}(l_0\ l_1\ v \cdots)] \rangle$ CALL
$\hookrightarrow \langle \mu, \alpha, J, E[\texttt{jp}(\lceil \texttt{call}, l_0, l_1, p_c \rceil, \alpha')] \rangle$
where
$\alpha' = (\texttt{asps}(\texttt{cobj}()) \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha)|_c$
$p_e = \textbf{fun}()\{\ \texttt{app/prim}\ \textbf{fun}(x \cdots)\{e\}\ l_0\ l_1\ v \cdots\ \}$
$p_c = \textbf{fun}()\{\ \texttt{app/prim}\ \textbf{fun}()\{\texttt{jp}(\lceil \texttt{exec}, l_0, l_1, p_e \rceil, \alpha')\}\}$

$\langle \mu, \alpha, J, E[\texttt{c/asp}\ k\ \textbf{fun}(x \cdots)\{e\}\ l_0\ l_1\ v \cdots] \rangle$ C/ASP
$\hookrightarrow \langle \mu, \alpha, J, E[\texttt{jp}(\lceil k, l_0, l_1, p \rceil, \alpha)] \rangle$
where $p = \textbf{fun}()\{\ \texttt{app/prim}\ \textbf{fun}(x \cdots)\{e\}\ l_0\ l_1\ v \cdots\ \}$

Figure 6.6: Join point creation.

6.4.2.3 Join Point Creation & Disposal

We change the NEW rule of λ_{JS} to account for the creation of `new` join points (Figure 6.6). The join point abstraction components are filled according to Figure 6.4. The primitive operation p is a thunk that returns a fresh reference to the newly-created object. Actual object creation is done using `new/prim`, an internal expression that performs creation without generating any join point. Note that the object value passed to `new/prim` is annotated with its initial aspect environment, α'. This environment is calculated as the order-preserving union ($\oplus$, See Appendix A) of three aspect environments: the ones deployed on the currently-executing object and function (obtained with `cobj` () and `cfun` (), respectively); and the stack aspect environment. Only aspects that propagate in newly-created objects are included in α'. The notation $\alpha|_d$ refers to the aspects in α whose `d` component is true.

To account for the creation of `call` and `exec` join points, we change the λ_{JS} evaluation rule for function application/method invocation as well. The new CALL rule generates a `call` join point whose components are filled according to Figure 6.4. The primitive operation p_c is a thunk that generates an `exec` join point when applied. The primitive operation of this `exec` join point, p_e, performs the actual function execution by means of `app/prim`, another internal expression that does not generate join points. Note that the `jp` expressions associated to both join points specify

$$\langle \mu, \alpha, j + J, E[\texttt{in-jp}\ (v)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[v] \rangle \qquad \textsc{OutJp}$$
$$\langle \mu, \alpha, j + J, E[\texttt{in-jp}\ (\textbf{err}\ v)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\textbf{err}\ v] \rangle \qquad \textsc{OutJp-Err}$$

Figure 6.7: Join point disposal.

that the stack aspect environment must change to α' when p_c or p_e are applied in order to reflect the propagation of aspects through the stack. This aspect environment is determined by taking the order-preserving union of three aspects environments: the ones deployed on the currently-executing object and function; and the stack aspect environment; and filtering the resulting environment along the `c` component (written $\alpha|_c$), which determines the aspects that should propagate on the call stack.

Rule C/ASP accounts for the creation of `pc-exec` and `adv-exec` join points. This rule matches a function application/method invocation, but receives a first argument (k) that specifies which join point must be generated. Because invocations of pointcuts and advices are implicit, C/ASP does not generate `call` join points. Join point attributes are filled according to Figure 6.4; the primitive operation p performs the pointcut/advice execution by means of `app/prim`, just like in the case of `exec` join points.

Once the computation underlying a join point is reduced to a value, the OUTJP rule gets rid of the join point and the `in-jp` expression (Figure 6.7). OUTJP-ERR does the same in the case of an error.

6.4.2.4 Weaving

We now turn to the semantics of aspect weaving, specified by the WEAVE rule (Figure 6.8). A `jp` expression reduces to an `in-jp` expression (to signal the fact that the upcoming computation is associated to a join point), and the join point is pushed onto the stack (we discuss the use of `swap` and α_s later below). The list of aspects in scope α' is calculated as the order-preserving union of the aspect environments of the object and function in context, and the aspects propagated through the stack.

The weaving process is based on evaluating the function returned by the W metafunction. W recurs on α' and returns a composed procedure whose structure reflects the way advice is going to be dispatched. The base case, $W[\![\epsilon]\!]$, corresponds to the execution of the primitive operation. Otherwise, for each aspect (b_c, b_d, l_{asp}) in the environment, W first applies its pointcut to the current

$\langle \mu, \alpha, J, E[\texttt{jp}(\lceil k, l_o, l_f, p \rceil, \alpha_p)] \rangle$ WEAVE
$\hookrightarrow \langle \mu, \alpha, J', E[\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\ W[\![\alpha']\!]_{\alpha_p, J'}, \epsilon))] \rangle$
where
$J' = \lceil k, l_o, l_f, p \rceil + J$
$\alpha_s = \epsilon$ if $k \in \{\texttt{pc-exec}, \texttt{adv-exec}\}, \alpha$ otherwise
$\alpha' = \texttt{asps}(\texttt{cobj}()) \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha_s$

$W[\![\epsilon]\!]_{\alpha, \lceil k, l_o, l_f, p \rceil + J} = \textbf{fun}()\{\texttt{swap}(\texttt{app/prim}\ p, \alpha)\}$
$W[\![\alpha_w + (b_c, b_d, l_{asp})]\!]_{\alpha, \lceil k, l_o, l_f, p \rceil + J} =$
$\texttt{app/prim}$
$\textbf{fun}(next)\{$
$\textbf{let}(pc = (\textbf{deref}\ l_{asp})[\text{"}pc\text{"}])$
$\textbf{if}(\texttt{c/asp pc-exec}\ (\textbf{deref}\ pc)[\text{"}code\text{"}]\ l_{asp}\ pc\ \boxed{j_p + J})\{$
$\textbf{let}(adv = (\textbf{deref}\ l_{asp})[\text{"}adv\text{"}])$
$\textbf{fun}()\{$
$\texttt{c/asp adv-exec}\ (\textbf{deref}\ adv)[\text{"}code\text{"}]\ l_{asp}\ adv\ \boxed{j_a + J}\ \}$
$\}\textbf{else}\{\ next\ \}$
$\}$
$W[\![\alpha_w]\!]_{\alpha, \lceil k, l_o, l_f, p \rceil + J, p}$
where $j_a = \lceil k, l_o, l_f, \textbf{fun}()\{\texttt{app/prim}\ next\} \rceil$
$j_p = \lceil k, l_o, l_f, \textbf{fun}()\{\textbf{err}\ \text{"pc cannot proceed"}\} \rceil$

Figure 6.8: Aspect weaving.

join point stack (which generates a `pc-exec` join point using the `c/asp` construct). If the pointcut matches, then W returns a function that applies the advice of l_{adv} (and generates an `adv-exec` join point). All this process is parameterized by the function to proceed with, $next$. In order to allow an advice to call **proceed** to trigger either the base computation or the next advice in the chain, rule WEAVE creates an auxiliary join point j_a whose p component is a thunk that applies $next$. To be complete, an auxiliary join point j_p is also created and passed to the pointcut; its p component triggers an error if **proceed** is called. Finally, If an aspect does not apply, then W simply returns $next$.

Primitive forms. The semantics of λ_{AS} use internal primitive forms `app/prim` and `new/prim`, described in Figure 6.9. `app/prim` is an application that does not trigger a join point: rule APPPRIM simply performs the classical β_v reduction. `app/prim` is used to perform the actual application of a

$$
\begin{array}{rrcl}
\mathit{Expr} & e & ::= & \ldots \mid \texttt{app/prim}\, e\, e \cdots \mid \texttt{new/prim}\, e \\
\mathit{EvalCtx} & E & ::= & \ldots \mid \texttt{app/prim}\, v \cdots\, E\, e \cdots \mid \texttt{new/prim}\, E
\end{array}
$$

$$
\begin{array}{ll}
\langle \mu, \alpha, J, E[\texttt{app/prim}\ \textbf{fun}(x \cdots)\{e\}\ v \cdots] \rangle & \textsc{AppPrim} \\
\hookrightarrow \langle \mu, \alpha, J, E[e[v \cdots / x \cdots]] \rangle & \\
\langle \mu, \alpha', J, E[\texttt{new/prim}\ o^{\alpha}] \rangle \hookrightarrow \langle \mu', \alpha', J, E[l] \rangle & \textsc{NewPrim} \\
\text{where} \quad l \notin \mathit{dom}(\mu) \text{ and } \mu' = \mu + (l \mapsto o^{\alpha}) &
\end{array}
$$

Figure 6.9: Primitive function application and object allocation.

function, as well as to hide "administrative" application, *i.e.* the initial application of the composed aspect chain, and its recursive applications. Similarly, `new/prim` allocates an object in the store and reduces to the corresponding location without producing a join point.[1]

Execution levels. The weaving semantics explained previously is insufficient, because any aspect language must take precautions with infinite regression. Indeed, if we omitted the use of `swap` and α_s in Figure 6.8, a λ_{AS} program would never terminate. Tanter addressed this issue with execution levels [74], which ensure that pointcut and advice computation by default always happen at a higher level than base computation, avoiding infinite loops such as those due to pointcuts matching against themselves. Recall that in λ_{AS}, pointcuts and advices are standard functions. With execution levels, pointcuts and advices are always evaluated at the level above the expression that generates a join point. When the last advice in the chain proceeds, execution shifts back to the original level in order to run the base computation.[2]

We introduce a simple modeling of execution levels, that does not require having to explicitly track the current execution level in the program configuration. Instead, we use the call stack with internal expressions so as to *swap* aspect environments and restore them when appropriate (Figure 6.10). Swapping per se is a very simple process: given an expression e and an aspect environment α', `swap` installs the aspect environment, and evaluates the expression (IN-SWAP).

[1] These primitive forms are necessary for the semantics to allow actual computation to happen. The fact that they are *internal* means that it is not necessary to protect them from untrusted aspects: they cannot be used by any user code, and cannot be advised since they do not produce join points. Recall that in a higher-order aspect language, the use of execution levels is key to supporting these primitive forms as internal only [74].

[2] Full-fledged execution levels include the possibility to explicitly shift execution up and down if needed, as well as to define level-capturing functions [74]. We do not include these advanced facilities in this work.

$$
\begin{array}{rrcl}
\mathit{Expr} & e & ::= & \ldots \mid \texttt{swap}(e,\alpha) \mid \texttt{in-swap}(e,\alpha) \\
\mathit{EvalCtx} & E & ::= & \ldots \mid \texttt{in-swap}(E,\alpha)
\end{array}
$$

$$
\begin{array}{ll}
\langle \mu, \alpha, J, E[\texttt{swap}(e,\alpha')] \rangle \hookrightarrow \langle \mu, \alpha', J, E[\texttt{in-swap}(e,\alpha)] \rangle & \textsc{In-Swap} \\
\langle \mu, \alpha', J, E[\texttt{in-swap}(v,\alpha)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[v] \rangle & \textsc{Out-Swap} \\
\langle \mu, \alpha', J, E[\texttt{in-swap}((\textbf{err}\ v),\alpha)] \rangle \hookrightarrow \langle \mu, \alpha, J, E[\textbf{err}\ v] \rangle & \textsc{Out-Swap-Err}
\end{array}
$$

Figure 6.10: Swapping aspect environments.

`in-swap` is used to restore the swapped aspect environment α when the expression is fully reduced (OUT-SWAP). Additionally, α_s is used to remove aspects in the stack aspect environment from scope when weaving `pc-exec` and `adv-exec` join points. This prevents the aspects deployed on the currently executing object/function from seeing their own activity. Note that this approach does support multiple levels of execution.

Weaving (Figure 6.8) uses `swap` exactly where the original levels semantics [74] uses **up** and **down** shifting. The whole weaving process is wrapped by a `swap`, so that the current aspect environment is swapped with an empty environment ϵ that represents the upper level environment . This environment is used to evaluate pointcuts and advices. Of course, the fact that the stack aspect environment starts empty does not prevent aspects that have been deployed in objects and functions from taking effect. If the last advice proceeds (the base case of W), aspect environments are swapped again, in order to restore the original environment to evaluate the base computation. The environment in which weaving is carried out is restored once the base computation has completed. Finally, when the whole weaving is complete, the original aspect environment is restored.

6.5 Properties of ModAC

In this section we state two theorems corresponding to the following properties of ModAC:

Basic effectiveness. ModAC is effective in absence of untrusted aspects. This means that restriction aspects are actually deployed on untrusted objects, see illegal resource accesses, and effectively prevent them.

Non-inhibition. ModAC with $\mathring{\text{R}}$ is effective even in presence of untrusted aspects. This means that

Å effectively prevents untrusted aspects from inhibiting protected aspects.

More precisely, we show the results for ModAC/λ_{AS}. The extension of these results to ModAC/AS (and other aspect languages) in Section 6.6. This section sketches the formal argument by describing the main intermediate steps. Each step includes an informal explanation, and a formal statement. The actual proofs, which rely on the operational semantics above, are provided in Appendix A.

First, we describe the properties that define basic effectiveness.

Definition 1 (Basic effectiveness)**.** *An implementation of ModAC is said to comply with basic effectiveness if the following properties are fulfilled:*

- Restrictions deployment. *Restrictions are deployed on all the corresponding objects before these objects can be used.*
- Restrictions scope. *A restriction aspect sees all the computation produced by the objects it is deployed on.*
- Restrictions effectiveness. *A restriction aspect always prevents the resource accesses it identifies.*

Theorem 1 (ModAC/λ_{AS} basic effectiveness)**.**
ModAC/λ_{AS} complies with basic effectiveness.

This theorem is a direct consequence of Lemmas 1, 2, and 3, exposed below, which address each property of basic effectiveness separately.

Lemma 1 states that any aspect (referenced by l_{depl}), in particular ACDeployer, deployed with scoping strategy (**false**,**true**) propagates to every new object in the store, and does so *in the first position* in the aspect environment of these objects. This ensures that l_{depl} sees all object creations in the application and gets a reference to these objects before any other entity. The only prerequisite is that l_{depl} is already deployed in the first position on all objects at a given point. This can be straightforwardly achieved in the bootstrapping process by exhaustively deploying ACDeployer on every object.[1]

Lemma 1 (Restrictions deployment)**.** *Let* $C = \langle \mu, \cdot, \cdot, \cdot \rangle$ *be a program configuration where* $\forall\ (l \mapsto o^{\alpha}) \in \mu$, $\alpha = (\boldsymbol{false}, \boldsymbol{true}, l_{depl}) + \alpha'$, *for some* α', *and* $l_{depl} \in dom(\mu)$. *If* $C \hookrightarrow \langle \mu', \cdot, \cdot, \cdot \rangle$, *then*

[1] Whenever an element of an entity (program configuration, join point, tuple, etc.) is not required, we use $\cdot$ as a wildcard.

$\forall\,(l \mapsto o^{\alpha}) \in \mu'$, $\alpha = (\textbf{\textit{false}}, \textbf{\textit{true}}, l_{depl}) + \alpha''$, *for some* α''.

Lemma 2 states that all aspects in the stack aspect environment deployed with c = **true**, propagate through the stack if the same level of execution is considered; *i.e.* the stack inspection algorithm is correctly implemented by means of scoping strategies.

Lemma 2 (Restrictions scope)**.** *Let* $C = \langle \cdot, \alpha, \cdot, \cdot \rangle$ *be a program configuration and* $\alpha_s = \alpha|_c$*. If* $C \hookrightarrow^* \langle \cdot, \alpha', \cdot, \cdot \rangle$*, and the sequence of reductions starts and ends at the same execution level, then* $\alpha_s \subseteq \alpha'$.

Lemma 3 states that if a restriction aspect R matches a join point j and does not proceed, then the primitive operation associated to j is not evaluated. Consequently a restriction aspect fulfills its role no matter in which position it is woven at the illegal resource access join point.

Lemma 3 (Restrictions effectiveness)**.** *Let* $C = \langle \mu, \cdot, J, E[e] \rangle$ *be a program configuration where* $J = \lceil \cdot, \cdot, \cdot, p \rceil + J'$*, for some* J'*,* $e = \texttt{app/prim}\, W[\![\alpha]\!]_{\cdot,J}$*,* $(\cdot, \cdot, l_R) \in \alpha$*, and* l_R *is a valid aspect reference in* μ *to a restriction aspect that matches* J *and does not proceed for* J*. If* $C \hookrightarrow^* \langle \cdot, \cdot, J, E[v] \rangle$*, for some* v*, then* p *is not applied in these reductions.*

Finally, we present the non-inhibition theorem. This theorem states that if the evaluation of a pointcut whose aspect has $\mathring{\text{R}}$ deployed on it reduces to a value, this value is either **false** or (**err** $\cdot$). This holds whenever the join point stack contains a join point whose target is in the set of protected aspects PA. Notice that the theorem implicitly permits the existence of other untrusted aspects trying to inhibit $\mathring{\text{R}}$ itself.

Theorem 2 (Non-inhibition)**.** *If* l_{asp} *is a valid aspect reference in* μ*,* $\mathring{R} \in \texttt{asps}(l_{asp})$*, and* $\lceil \cdot, s, \cdot, \cdot \rceil \in J$*; where* $s \in PA$*, then:*
If $\langle \mu, \alpha, J, E[\texttt{jp}(\lceil \texttt{pc-exec}, l_{asp}, \cdot, \cdot \rceil, \cdot)] \rangle \hookrightarrow^* \langle \cdot, \alpha, J, E[v] \rangle$*, then* $v =$ ***false*** *or* $v = (\textbf{\textit{err}}\ \cdot)$.

6.6 Discussion

We discuss how to extend our results from λ_{AS} to full-fledged AspectScript, and the requirements for a general-purpose aspect language to securely support ModAC.

From λ_{AS} ***to AspectScript.*** Due to desugaring, results obtained in λ_{JS} do not immediately apply

to JavaScript [40]. This is because desugaring introduces new behavior that was not present in the original code. When going from λ_{AS} to AspectScript, the theorems remain valid because they are based on the aspect-oriented features of the language, which have no relation to the desugaring process. However, access control aspects can be led to behave incorrectly if they use "exploitable" features that introduce holes upon desugaring. For example, consider a slight modification of the pointcut of the netRestriction aspect in order to allow communication with safe.cl:

```
function(jp){ return /* same as before */ && !(jp.args[0] == "safe.cl"); }
```

The equality operator == forces both operands to be of the same type [44]. For this reason, jp.args[0] is transformed to a string by an invocation of toString. The problem is that this extra method call opens the opportunity for bypassing access control:

```
var req = netService.newRequest(
        { t: 0, toString: function(){return ["safe.cl", "evil.com"][this.t++]}});
```

The toString method of the argument to newRequest returns "safe.cl" the first time it is invoked (in the pointcut of netRestriction) and "evil.com" the second time (in the body of newRequest).

In order to avoid such holes, the first possibility is to simply avoid using exploitable features in the definition of restriction aspects. For instance, it is safe to use reference equality === because it does not perform any kind of type conversion [44] (notice that all restriction aspects defined in this work follow this guideline). A less drastic solution is to permit the use of exploitable features, but to carefully examine access control aspects in order to check if their *particular usage* of the feature is safe. For example, the equality operator == is safe if both operands are of the same runtime type! As detailed by Guha *et al.*, this checking can be automated by a specialized type system [40].

Finally, AspectScript uses a scoping strategy acs, which supports privileged execution and capturable permission contexts; acs is expressed with propagation functions, c and d. We made a simplification in λ_{AS} by supporting only constant boolean propagation values. As we said in Section 6.4.2.2, this simplification does not affect our results. In fact, supporting propagation functions only requires that $\mathring{\text{R}}$ prevents inhibition of these functions; this is achieved by extending the pointcut of $\mathring{\text{R}}$:

```
function(jp){return ... || cflow(function(jp){return acs.contains(jp.fun);});}
```

This way, $\mathring{\text{R}}$ also inhibits untrusted aspects in the control flow of acs components. Theorem 2 and its proof must be reformulated accordingly, but this is direct.

Aspect languages for secure ModAC. This paper focuses on AspectScript to be as close as possible to our practical implementation, ZAC. Still, both ModAC and the approach for securing it using Ř are independent of AspectScript. They can be realized in any aspect language, provided it meets certain key conditions. First of all, the language must support scoping strategies, or an equivalently expressive scoping mechanism. Per-object aspects are only necessary if one wants to provide per-object access control. Execution levels are necessary to avoid infinite loops whenever pointcut and/or advice execution join points are exposed to weaving; in order to control implicit inhibition, Ř relies on matching pointcut execution join points.

A crucial point in ModAC that is directly informed by the formal framework and explicitly used in Lemma 1 is related to aspect precedence: ACDeployer must always be the aspect with least precedence in the aspect environment to be woven at a `new` join point. This allows ACDeployer to deploy restriction aspects on objects before they get a chance to execute any piece of code. The semantics of λ_{AS} ensures this premise because per-object aspects are "engrained" within the object following the semantics of dynamically-deployed aspects in AspectScheme [26]. In AspectScheme this is a design decision; here it is not—it is a *requirement*. If an aspect language uses a different approach to ordering aspects, or permits to undeploy aspects, then it must provide a mechanism to guarantee the above invariant related to the presence and position of ACDeployer. For example, in AspectJ [49], aspects cannot be undeployed, but manual ordering is provided. Therefore, some mechanism must be added, as in AOPS [23]. On a related note, it is necessary that ACDeployer can deploy the restrictions on a newly-created object before any code is run on behalf of this object. In λ_{AS}, this is obtained thanks to the desugaring, which creates an empty object and then calls an initializer. In AspectJ, this can be achieved thanks to the pre-initialization join points. If an aspect language does not exhibit this specific event of an object life time, then it is not possible to guarantee that restrictions see all the computation of untrusted objects.

6.7 Related Work

We now discuss a number of related approaches. To the best of our knowledge, AOPS is the only approach that supports untrusted aspects while preventing inhibitions of access control aspects.

Limiting the effects of advice. A number of static reasoning approaches deal with ensuring that advice cannot have unwanted effect on the base program (*e.g.* Harmless Advice [21], EffectiveAdvice [58], Translucid Contracts [11]). These proposals focus on control flow and side effects, and

can therefore express the fact some aspects cannot skip proceed. However, inhibition of access control as dealt with in this work requires more fine-grained control, since it limits untrusted aspects only on well-defined join points, leaving them otherwise unrestricted.

Treatment of permission contexts. Caromel and Vayssière addressed the issue of correctly handling permission contexts in the presence of metaobjects [18]. The issue is to ensure that the permission context at the base level does not affect that of the metalevel, and vice-versa. The proposed solution relies on capturing the permission context when jumping to the metalevel, and restoring it when going back to the base level. Because permission contexts are part of aspect environments in ModAC, we generalize this approach to deal with aspect environments (using `swap` and `in-swap`); also, our execution-level based approach properly deals with proceed.

Preventing access control inhibition. The aspect-oriented permission system (AOPS) [23] is the most related approach; it uses history-based access control (HBAC) [1], in which the decision of allowing access to a sensitive resource is taken based on *all* the entities that have participated in the execution trace. This characteristic makes HBAC a good alternative for discovering interferences produced by untrusted aspects. As discussed in Section 6.3.2, AOPS sacrifices two of our design goals: the aspect language semantics is customized to prevent weaving of crucial elements of the access control architecture (G1), thereby impeding even trusted aspects to apply at these points (G3). This being said, history-based access control is more expressive than stack-based access control. Extending or adapting our approach to HBAC is a potentially fruitful perspective.

6.8 Summary

In Chapter 3 we presented ModAC, our approach for the modularization of access control, including privileged execution and first-class permission contexts. In this chapter we showed two ModAC properties related to security and usability: untrusted aspects cannot inhibit access control, and trusted aspects are able to see any join point. The approach relies on defining $\mathring{\text{R}}$, a self-protecting restriction aspect in ModAC. $\mathring{\text{R}}$ is in charge of ensuring *non-inhibition* of access control. We define λ_{AS}, a core calculus for AspectScript, and use it for stating and proving the properties of ModAC. Crucially, the language must provide some guarantee with respect to aspect precedence.

Availability. The proofs are available in Appendix A. This work is implemented in the ZAC library for AspectScript. Also, the executable semantics of λ_{AS} are implemented in PLT Redex [31]. Both artifacts are available online http://users.dcc.uchile.cl/~rtoledo/ring/.

Chapter 7

Conclusions

In this chapter we first review the main contributions of our research. Then, we discuss some of the perspectives for future work.

7.1 Contributions

The main contributions of this thesis are twofold:

- ModAC: Full aspectization of access control
- Formal guarantee of ModAC security properties

We detail each contribution in the following two subsections.

7.1.1 ModAC: Full Aspectization of Access Control

In Chapter 3 we described how access control, including advanced features like privileged execution and first-class permission contexts, can be modularized as an aspect. Then, in Chapter 6 we described how this modularization can be considered, under our own definition, *full*:

- The base language must be completely oblivious to access control

- Untrusted aspects must not inhibit protected aspects, but are otherwise free to advise any join points
- Trusted aspects should be able to advise any join point

Our approach, ModAC for Modular Access Control, is based on per-object deployment of restriction aspects on untrusted entities. Restriction aspects identify and forbid a potentially dangerous action as soon as they see it. Restriction aspects only see dangerous actions caused by untrusted entities since they are deployed with the access control scoping strategy, in charge of controlling when restriction aspects should propagate.

In order to secure ModAC under the potentially dangerous presence of malicious entities, we distinguish two kinds of inhibition:

- Explicit inhibition, derived from the explicit manipulation of data structures related to access control
- Implicit inhibition, derived from intervention in the weaving of restriction aspects or the ACDeployer aspect, and also derived from intervention in the evaluation of the access control scoping strategy

Thanks to the elegant notion of restriction aspects, both kinds of inhibition are prevented in the same manner. Explicit inhibition is prevented by the definition of dedicated restriction aspects; and implicit inhibition is prevented by $\mathring{\text{R}}$, a self-protecting restriction aspect in charge of ruling all other aspects.

7.1.2 Formal Guarantee of ModAC Security Properties

In Chapter 6 we formally prove two security properties of ModAC:

- Basic effectiveness, meaning that restriction aspects are actually deployed on untrusted objects, see illegal resource accesses, and effectively prevent them
- Non-inhibition, meaning that when $\mathring{\text{R}}$ is deployed on untrusted objects, they are unable to interfere with restriction aspects, the access control scoping strategy, and the ACDeployer aspect.

In order to prove these properties, we developed AspectScript, an aspect-oriented extension to JavaScript, presented in Chapter 4. Then, in Chapter 6, we presented the semantics of AspectScript, expressed in the core calculus λ_{AS}.

Additional Contributions. Apart from the two main contributions presented above, we developed artifacts that helped us to test and experiment with our ideas. Among the most relevant ones we can find the following:

- List of restriction aspects equivalent to the Java access control architecture (Chapter 3)
- ModAC/AS & ZAC prototypes (Chapter 5)
- PLT Redex semantics for λ_{AS} (Chapter 6)

7.2 Perspectives

This thesis work focused exclusively on describing a way to modularly define access control, and then, on proving that it was actually a safe alternative to use in a real language. In this section we briefly mention some of the topics we left aside and that we consider worth exploring.

7.2.1 Performance Optimization

One of the most typical trade-offs that must be faced in the quest for general-purpose solutions is performance. In this thesis, we focused on having the semantics right in the first place, leaving aside most of the performance issues (although we developed some minor optimizations to solve the biggest sources of performance degradation). This fact is reflected in Sections 4.3.3 and 5.3.7, where we presented some preliminary performance numbers, and the obvious conclusion is that the runtime performance of AspectScript should be improved.

As mentioned in Section 3.6.3, there is a major performance optimization opportunity based on partial evaluation [45]. This can be devised since the access control scoping strategy is always the same for every restriction aspect, and also it is known in advance. Actually, both functions, `c` and `d` can be statically evaluated (the only condition that must be left for evaluation at runtime is determining whether a call to `doPriviliged` is a self call or not). In the case of Java, the AspectBench Compiler [10] for AspectJ is targeted to make it easy to extend and optimize new language features.

Implementing the semantics of scoping strategies in this compiler would facilitate its performance optimization.

7.2.2 Mechanic Proofs

In Chapter 6 we formally prove the basic effectiveness and non-inhibition properties over the core calculus λ_{AS}. However, these proofs were almost completely conducted in a manual manner, and although all claims were rigorously checked by us and by several reviewers, there is a possibility that a mistake was overlooked.

In order to completely eliminate this possibility, formal proof management systems like Coq [12] can be used. Although each step in the proof must be manually and explicitly specified, the biggest advantage of these systems is that they do not accept an incorrect step. Consequently, once a proof for a theorem is obtained, it can be ascertained that the theorem is true.

7.2.3 Distributed Security

Access control is complicated enough when considering just one system. In this thesis work we explored how to safely load potentially malicious *code* by applying restrictions to it. However, there is another dimension in which two separate systems share remote object references and execute code not in the same host, but in another one.

Scoping strategies have been shown to be useful in a distributed context [75]. In addition, most restriction aspects are, by construction, stateless, which facilitates their (de-) serialization in order to be transfered to different hosts. These two conditions opens the possibility for ModAC to be extended to support modular access control in a distributed setting.

List of Figures

Bibliography

[1] M. Abadi and C. Fournet. Access control based on execution history. *In Proceedings of the 10th annual Network and Distributed System Security Symposium*, pages 107–121, 2003. 46, 101

[2] AdSafe. http://www.adsafe.org. 77

[3] Ajaxpect. A JavaScript framework for aspect-oriented programming. http://code.google.com/p/ajaxpect/. 62

[4] D. Alhadidi, A. Boukhtouta, N. Belblidia, M. Debbabi, and P. Bhattacharya. The dataflow pointcut: a formal and practical framework. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 15–26, Charlottesville, Virginia, USA, 2009. ACM. 9

[5] *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, Apr. 2008. ACM Press. 109, 112

[6] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press. 109, 111, 112

[7] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, Feb. 2006. 7, 49, 53

[8] AspectJS. A function-call framework in JavaScript. http://www.aspectjs.com/. 62

[9] AspectJS. A JavaScript framework for aspect-oriented programming. http://zer0.free.fr/aspectjs/. 62

[10] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag, 2006. 84, 104

[11] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, Porto de Galinhas, Brazil, Mar. 2011. ACM Press. 100

[12] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. 105

[13] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition, 2006. 39, 55

[14] D. Box and C. Sells. *Essential .NET: The common language runtime*, volume 1. Addison-Wesley, Nov. 2002. 2, 4

[15] D. Brewer and M. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, page 206, Los Alamitos, CA, USA, 1989. IEEE Computer Society. 40

[16] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. EJFlow: Taming exceptional control flows in aspect-oriented programming. In AOSD 2008 [5], pages 72–83. 38

[17] Caja. http://code.google.com/p/google-caja. 77

[18] D. Caromel and J. Vayssière. A security framework for reflective Java applications. *Software: Practice and Experience*, 33(9):821–846, 2003. 101

[19] Cerny. A javascript framework for method-call interception. http://www.cerny-online.com/cerny.js/. 62

[20] C. Clifton and G. T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:312–374, 2006. 89

[21] D. S. Dantas and D. Walker. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 383–396, Charleston, South Carolina, USA, Jan. 2006. ACM Press. 100

[22] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008. 63

[23] W. De Borger, B. De Win, B. Lagaisse, and W. Joosen. A permission system for secure AOP. In AOSD 2010 [6], pages 205–216. 81, 83, 100, 101

[24] B. De Win, B. Vanhaute, and B. De Decker. Security through aspect-oriented programming. In B. De Decker, F. Piessens, J. Smits, and E. Van Herreweghen, editors, *Advances in Network and Distributed Systems Security*, pages 125–138. Kluwer Academic Publishers, 2001. 10

[25] E. W. Dijkstra. The structure of the THE multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968. 1

[26] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006. 7, 49, 51, 53, 60, 62, 89, 100

[27] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001. 78

[28] U. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 246–255, 2000. 10, 26

[29] FBJS. http://wiki.developers.facebook.com/index.php/FBJS. 77

[30] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991. 13

[31] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. 101

[32] D. Ferraiolo and R. Kuhn. Role-Based access control. *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992. 32, 68

[33] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranho, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–162, Portland, Oregon, USA, 2006. ACM. 38

[34] J. Fischer, D. Marino, R. Majumdar, and T. Millstein. Fine-Grained access control with Object-Sensitive roles. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *Lecture Notes in Computer Science*, pages 173–194, Genova, Italy, July 2009. Springer Berlin / Heidelberg. 44

[35] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(3):360 – 399, 2003. 5

[36] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 348–355, Aug. 1984. 15

[37] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003. 11, 14, 15, 40

[38] L. Gong, M. Mueller, H. Prafullchandra, R. Schemers, and S. Microsystems. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. *Proceedings of the USENIX Symposium for Secure Systems*, 1997. 11, 15

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3rd edition*. Addison-Wesley, 2005. 2, 4

[40] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In T. D'Hondt, editor, *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, number 6183 in Lecture Notes in Computer Science, pages 126–150, Maribor, Slovenia, June 2010. Springer-Verlag. 79, 87, 88, 99

[41] N. Hardy. The confused deputy. *SIGOPS Operating Systems Review*, 22(4):36–38, 1988. 12

[42] M. Huang, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. In *AOSD Technologies for Application-Level Security*, 2004. 10

[43] Humax. A JavaScript framework for aspect-oriented programming. http://humax.sourceforge.net/. 62

[44] E. International. *ECMAScript Language Specification. ECMA-262*. Ecma International, 5 edition, Apr. 2009. 99

[45] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993. 104

[46] jQuery. A JavaScript library to manage event handling, animating, and Ajax interactions for the Web development. http://jquery.com/. 60, 62, 76

[47] JSON. http://www.json.org. 73

[48] G. Karjoth. An operational semantics of Java 2 access control. In *Computer Security Foundations Workshop, IEEE*, page 224, Washington, DC, USA, 2000. IEEE Computer Society. 15

[49] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag. 7, 13, 23, 50, 100

[50] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996. 1

[51] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. Language Objective Caml: Caml supports functional, imperative, and object-oriented programming styles. http://caml.inria.fr/. 63

[52] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121, Nov. 2003. 9

[53] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003. 6, 51

[54] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 320–330, Tallinn, Estonia, 2005. ACM. 63

[55] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. 63

[56] A. Mourad, M. Laverdire, and M. Debbabi. An aspect-oriented approach for the systematic security hardening of code. *Computers & Security*, 27(3-4):101–114, June 2008. 10

[57] A. Mourad, A. Soeanu, M. Laverdire, and M. Debbabi. New aspect-oriented constructs for security hardening concerns. *Computers & Security*, 28(6):341–358, Sept. 2009. 9

[58] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [6], pages 109–120. 100

[59] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972. 1

[60] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified Access-Control and Information-Flow security model. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 149–163. IEEE Computer Society, 2007. 44

[61] Prototype. A JavaScript library that aims to ease development of dynamic Web applications. http://www.prototypejs.org/. 62

[62] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 155–179, Paphos, Cyprus, July 2008. Springer-Verlag. 50, 51

[63] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007. 77

[64] G. Richards, C. Hammer, F. Zappa Nardelli, S. Jagannathan, and J. Vitek. Flexible access control for javascript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 305–322, New York, NY, USA, 2013. ACM. 77

[65] P. Samarati and S. D. C. di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Berlin / Heidelberg, London, UK, 2001. 4, 14

[66] N. Schärli, A. Black, and S. Ducasse. Object-oriented encapsulation for dynamically-typed languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 130–149, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11). 47

[67] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. 26

[68] J. Stamey, B. Saunders, and S. Blanchard. The aspect-oriented Web. In *Proceedings of the 23rd annual international Conference on Design of Communication: documenting & designing for pervasive information*, pages 89–95, Coventry, United Kingdom, 2005. ACM. 62

[69] Sun Microsystems Inc. Java security architecture. http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-specTOC.fm.html. 14, 15, 17, 19

[70] P. Sowikowski and K. Zieliski. Comparison study of aspect-oriented and container managed security. In *Proceedings of the Workshop on Analysis of Aspect Oriented Software*, Germany, 2003. 10

[71] É. Tanter. Controlling aspect reentrancy. *Journal of Universal Computer Science*, 14(21):3498–3516, 2008. 49, 55

[72] É. Tanter. Expressive scoping of dynamically-deployed aspects. In AOSD 2008 [5], pages 168–179. 8, 9, 13, 31, 42, 43, 46, 49

[73] É. Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*, pages 3–14, Orlando, FL, USA, Oct. 2009. ACM Press. 8, 9, 31, 42, 49

[74] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [6], pages 37–48. 39, 55, 56, 79, 87, 89, 95, 96

[75] É. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. Expressive scoping of distributed aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 27–38, Charlottesville, Virginia, USA, Mar. 2009. ACM Press. 9, 13, 42, 105

[76] É. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. Scoping strategies for distributed aspects. *Science of Computer Programming*, 75(12):1235–1261, Dec. 2010. 90

[77] P. L. Tarr, H. L. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999. 1

[78] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [6], pages 13–24. 49

[79] R. Toledo, A. Núñez, É. Tanter, and J. Noyé. Aspectizing Java access control. *IEEE Transactions on Software Engineering*, 38(1):101–117, Jan./Feb. 2012. 11

[80] R. Toledo and É. Tanter. Access control in JavaScript. *IEEE Software*, 28(5):76–84, Sept./Oct. 2011. 65

[81] R. Toledo and É. Tanter. Secure and modular access control with aspects. In J. Kinzle, editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*, pages 157–170, Fukuoka, Japan, Mar. 2013. ACM Press. 65, 79

[82] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In M. Akşit, editor, *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 158–167, Boston, MA, USA, Mar. 2003. ACM Press. 49

[83] B. Vanhaute, B. D. Decker, B. D. Win, and D. Decker. Building frameworks in AspectJ. *Workshop on Advanced Separation of Concerns (ECOOP)*, pages 1–6, 2001. 10

[84] D. Wallach, A. Appel, and E. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):341–378, 2000. 26, 48

[85] D. Wallach and E. Felten. Understanding Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, 1998. 26, 48

[86] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, Sept. 2004. 6

[87] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: aspect-oriented JavaScript programming framework for web development. In *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 31–36, Charlottesville, Virginia, USA, 2009. ACM. 62

[88] I. Welch and R. Stroud. Supporting real world security models in Java. In *Distributed Computing Systems, 1999. Proceedings. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 155–159, 1999. 2

[89] B. D. Win, W. Joosen, and F. Piessens. AOSD security: A practical assessment. *Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2003:1–6, 2003. 10

[90] B. D. Win, W. Joosen, and F. Piessens. Developing secure applications through Aspect-Oriented programming. In *Aspect-Oriented Software Development*, pages 633—650. Addison-Wesley Professional, Oct. 2004. 10

[91] B. D. Win, B. Vanhaute, and B. D. Decker. How aspect-oriented programming can help to build secure software. *Informatica 26*, 2:141–149, 2002. 10

[92] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Journal of Information and Computation*, 115(1):38–94, Nov. 1994. 88

Appendix A

ModAC Proofs

Here we present the proofs of the theorems stated in Chapter 6.

Conventions

There are some conventions we use:

- When the actual value of an element is not relevant, we omit it and use a $\cdot$. For instance, when we present a program configuration and the only interesting element is the store, we just use "$\langle \mu, \cdot, \cdot, \cdot \rangle$" instead of "$\langle \mu, \alpha, J, e \rangle$, for some α, J, and e".
- We use the notation $o[p]$ to refer to the property p of o. The value of the property is determined as follows:

$$o[p] = \begin{cases} v_i \text{ if } str_i = p, \text{ where } o = \{str_i : v_i\} \\ otherwise \end{cases}$$

Semantic elements not present in Chapter 6

The order-preserving union $\oplus$ used in the semantics:

$$(b,b,l)_1 + \ldots + (b,b,l)_n \oplus (b,b,l)_{n+1} + \ldots + (b,b,l)_{n+m} =$$
$$(b,b,l)_1 \oplus \ldots \oplus (b,b,l)_n \oplus (b,b,l)_{n+1} \oplus \ldots \oplus (b,b,l)_{n+m}$$

where

$$(b,b,l)_1 + \ldots + (b,b,l)_n \oplus (b,b,l)_k =$$
$$\begin{cases} (b,b,l)_1 + \ldots + (b,b,l)_n & \text{if } (b,b,l)_k \in (b,b,l)_1 + \ldots + (b,b,l)_n \\ (b,b,l)_1 + \ldots + (b,b,l)_n + (b,b,l)_k & \text{otherwise} \end{cases}$$

The stack introspection primitives:

$$\begin{array}{rcll} \textbf{kind}(\boxed{\lceil k, l_o, l_f, p \rceil} + J) & = & k & \textsc{Kind} \\ \textbf{tobj}(\boxed{\lceil k, l_o, l_f, p \rceil} + J) & = & l_o & \textsc{TObj} \\ \textbf{tfun}(\boxed{\lceil k, l_o, l_f, p \rceil} + J) & = & l_f & \textsc{TFun} \\ \textbf{proceed}(\boxed{\lceil k, l_o, l_f, p \rceil} + J) & = & (\texttt{app/prim}\ p) & \textsc{Proceed} \\ \textbf{parent}(\boxed{j + \epsilon}) & = & \textbf{null} & \textsc{NoParent} \\ \textbf{parent}(\boxed{j + J}) & = & \boxed{J} & \textsc{Parent} \end{array}$$

Basic effectiveness

Theorem 1 (ModAC/λ_{AS} basic effectiveness)**.** *ModAC/λ_{AS} complies with basic effectiveness.*

As stated in Chapter 6, this theorem is a direct consequence of Lemmas 1, 2, and 3, exposed below.

Lemma 1 (Restrictions deployment)**.** *Let $C = \langle \mu, \cdot, \cdot, \cdot \rangle$ be a program configuration where $\forall\ (l \mapsto o^{\alpha}) \in \mu$, $\alpha = (\textbf{\textit{false}}, \textbf{\textit{true}}, l_{depl}) + \alpha'$, for some α', and $l_{depl} \in dom(\mu)$. If $C \hookrightarrow \langle \mu', \cdot, \cdot, \cdot \rangle$, then $\forall\ (l \mapsto o^{\alpha}) \in \mu'$, $\alpha = (\textbf{\textit{false}}, \textbf{\textit{true}}, l_{depl}) + \alpha''$, for some α''.*

Proof. By induction on the reduction relation.

We analyze only the rules that actually change the store. For the remaining rules, the lemma trivially holds.

- (NEWPRIM) Rule NEWPRIM adds $(l \mapsto o^{\alpha})$ to the store, where l is a fresh location, and o is the referenced object. The value of α is calculated as follows:

$$
\begin{aligned}
\alpha \quad &= \quad (\texttt{asps}(\texttt{cobj}()) \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha_n)|_d \text{ for some } \alpha_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(rule NEW)} \\
&= \quad ((\textbf{false}, \textbf{true}, l_{depl}) + \alpha' \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha_n)|_d, \text{ for some } \alpha' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by I.H.)} \\
&= \quad ((\textbf{false}, \textbf{true}, l_{depl}) + \alpha'')|_d, \text{ for } \alpha'' = \alpha' \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(because the } \oplus \text{ operator is order-preserving)} \\
&= \quad (\textbf{false}, \textbf{true}, l_{depl}) + \alpha''', \text{ for } \alpha''' = \alpha''|_d \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by the definition of } |_d)
\end{aligned}
$$

- (DEPLOYON) Rule DEPLOYON modifies the store by replacing a mapping $(l \mapsto o^{\alpha})$ with $(l \mapsto o^{\alpha'})$, where α' is calculated as follows:

$$
\begin{aligned}
\alpha' \quad &= \quad \alpha + (b_c, b_d, l_{asp}), \text{ for some } b_c, b_d, \text{ and } l_{asp} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(rule DEPLOYON)} \\
&= \quad (\textbf{false}, \textbf{true}, l_{depl}) + \alpha'' + (b_c, b_d, l_{asp}), \text{ for some } \alpha'' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by I.H.)} \\
&= \quad (\textbf{false}, \textbf{true}, l_{depl}) + \alpha''', \text{ for } \alpha''' = \alpha'' + (b_c, b_d, l_{asp})
\end{aligned}
$$

□

For definition 1 we subscript the IN-SWAP rule to differentiate each one of its two uses. We use IN-SWAP$_w$ to refer to the use of `swap` in rule WEAVE, and IN-SWAP$_p$ to refer to the use of `swap` in the base case of the W metafunction.

Definition 1 (Same execution level)**.** *A sequence of transitions starts and ends at the same execution level if one of the three following conditions holds:*

- *Rules* IN-SWAP$_w$*,* IN-SWAP$_p$*, and* OUT-SWAP *do not appear*
- *Among the three rules,* IN-SWAP$_w$ *is the first one to appear, and rule* OUT-SWAP *is the last one; and the sequence of transitions between them starts and ends at the same execution level*
- *Among the three rules,* IN-SWAP$_w$ *is the first one to appear, and rule* IN-SWAP$_p$ *is the last one; and the sequence of transitions between them starts and ends at the same execution level*

Lemma 2 (Restrictions Scope)**.** *Let* $C = \langle \cdot, \alpha, \cdot, \cdot \rangle$ *be a program configuration where* $\alpha_s = \alpha|_c$*. If* $C \hookrightarrow^* \langle \cdot, \alpha', \cdot, \cdot \rangle$*, and the sequence of reductions starts and ends at the same execution level, then* $\alpha_s \subseteq \alpha'$*.*

Proof. By induction on the reduction relation.

We analyze only the rules that actually change the stack aspect environment. For the remaining rules, the lemma trivially holds.

- (IN-SWAP$_w$) A balanced swap transition cannot end with this rule, so the lemma is vacuously true.

- (IN-SWAP$_p$) IN-SWAP$_p$ changes the stack aspect environment to some α' (the first parameter to the W meta-function). This α' is α_p in rule WEAVE, which is calculated differently according to the kind of join point being weaved:
  - (`new`, `pc-exec`, and `adv-exec`)

    α' = α, where α is the current aspect environment
    (rules NEW and CALLASP)

    The key is that α is calculated before the first IN-SWAP$_w$, therefore,
    the sequence of reductions is at the same level of execution at that point.

    $\supseteq$ α_s
    (by I.H.)

  - (`call` and `exec`)

    α' = $(\texttt{asps}(\texttt{cobj}()) \oplus \texttt{asps}(\texttt{cfun}()) \oplus \alpha)|_c$
    where α is the current aspect environment
    (rule CALL)

    $\supseteq$ $\alpha|_c$

    Again, the key is that α is calculated before the first IN-SWAP$_w$, therefore,
    the sequence of reductions is at the same level of execution at that point.

    $\supseteq$ $\alpha_s|_c$
    (by applying $(\cdot)|_c$ to the I.H.)

    = α_s
    (because $\alpha_s = \alpha_s|_c$)

□

Definition 2 (Valid Aspect Reference)**.** *A reference l is a valid aspect reference in a store μ iff $\mu(l)[pointcut] = l_{pc}$, $\mu(l)[advice] = l_{adv}$, $\mu(l_{pc})[code] =$ **fun**$(this, fthis, \ldots)\{\ldots\}$, and $\mu(l_{adv})[code] =$ **fun**$(this, fthis, \ldots)\{\ldots\}$.*

Definition 3 (Matching)**.** *A valid aspect reference l_{asp} (with pointcut l_{pc}) in μ matches a join point stack J iff $\langle \mu, \alpha_J, J, E[\texttt{jp}\ \lceil \texttt{pc-exec}, l_{asp}, [J], \cdot \rceil l_{asp} l_{pc}] \rangle \hookrightarrow^* \langle \mu', \alpha_J, J, E[\textbf{\textit{true}}] \rangle$, for some μ', and α_J.*

Definition 4 (Not Proceeding)**.** *A valid aspect reference l_{asp} (with advice l_{adv}) in μ is said to not proceed for J iff in the following sequence of reductions:*
$\langle \mu, \alpha_J, J, E[\texttt{jp}\ \lceil \texttt{adv-exec}, l_{asp}, [p, J], \cdot \rceil l_{asp} l_{adv}] \rangle \hookrightarrow^* \langle \mu', \alpha_J, J, E[v] \rangle$, p is not evaluated; for some μ', α_J, and v.

Lemma 3 (Restrictions Effectiveness)**.** *Let $C = \langle \mu, \cdot, J, E[e] \rangle$ be a program configuration where $J = \lceil \cdot, \cdot, \cdot, p \rceil + J'$, for some J', $e =$ `app/prim` $W[\![\alpha]\!]_{\cdot,J}$, $(\cdot, \cdot, l_R) \in \alpha$, and l_R is a valid aspect reference in μ to a restriction aspect that matches J and does not proceed for J. If $C \hookrightarrow^* \langle \cdot, \cdot, J, E[v] \rangle$, for some v, then p is not applied in these reductions.*

Proof. By analysis of the result of $W[\![\alpha]\!]_{\cdot,J}$.

The result of $W[\![\alpha]\!]_{\cdot,J}$ is a set of weaving functions that weave the different aspects in α. Each weaving function follows this shape:

$$
\begin{array}{l}
\textbf{fun}(next)\{ \\
\quad \textbf{let}(pc = (\textbf{deref}\ l_{asp})[\text{"}pc\text{"}]) \\
\quad \textbf{if}(\texttt{c/asp pc-exec}\ (\textbf{deref}\ pc)[\text{"}code\text{"}]\ l_{asp}\ pc\ \boxed{j_p + J}) \{ \\
\quad\quad \textbf{let}(adv = (\textbf{deref}\ l_{asp})[\text{"}adv\text{"}]) \\
\quad\quad \textbf{fun}()\{\ \texttt{c/asp adv-exec}\ (\textbf{deref}\ adv)[\text{"}code\text{"}]\ l_{asp}\ adv\ \boxed{j_a + J}\ \} \\
\quad \}\textbf{else}\{\ next\ \} \\
\}
\end{array}
$$

Where l_{asp} is the aspect being woven, $j_p = \lceil k, l_o, l_f, \textbf{fun}()\{\textbf{err}\ \text{"pc cannot proceed"}\}\rceil$, $j_a = \lceil k, l_o, l_f, \textbf{fun}()\{\texttt{app/prim}\ next\}\rceil$, and $next$ is the weaving function that corresponds to the next woven aspect. When there are no more aspect to weave (case $W[\![\epsilon]\!]_{\cdot,J}$), $next$ is the application of the original primitive operation:

$$\textbf{fun}()\{\ \texttt{app/prim}\ p\ \} \qquad\qquad (\texttt{swap}\ \text{rule omitted})$$

By inspection, there are only two ways for an aspect to cause the application of the next weaving function $next$:

- By not matching J, in which case the **else** branch is executed and hence the weaving function itself evaluates to $next$ which is, later on, applied.
- By matching J, and using **proceed**(J) in the execution of the advice, which applies the p attribute (rule PROCEED) of the j_{adv} join point abstraction , which in this case, is $next$.

In consequence, in order for p to be applied, all the aspects that match J must also proceed for J. Since l_R matches J but does not proceed for J, p is not applied. □

Non-inhibition

Here we prove our non-inhibition theorem. We start by proving a simpler property and then proceed with the full theorem.

Definition 5 (Protected aspect in the stack)**.** *It is said that a protected aspect is in the stack iff in the current program configuration $\langle \cdot, \cdot, J, \cdot \rangle, \exists \lceil \cdot, s, \cdot, \cdot \rceil \in J$, where $s \in PA$, the set of protected aspects.*

First, we assume that the implementation of $\mathring{\text{R}}$ is well-defined and that for any join point, the evaluation of its pointcut and advice will not diverge (but can produce an error for some reason) if no other aspect intervene in the computation.

Based on this, we prove the following property: lets say that $\mathring{\text{R}}$ requires n join points to achieve its purpose, if at least one of these join points is seen by other aspects for every application of $\mathring{\text{R}}$, then the computation ends with an error or diverges. Note that among these pointcuts are uses of sensible elements like cflow and the protectedAspects

set. The property we are about to prove holds for any implementation of $\mathring{\text{R}}$.

(From now on, in order to simplify the notation, we use $\mathring{\text{R}}$ to refer to a reference of $\mathring{\text{R}}$, and U to refer to a reference to any untrusted aspect)

Lemma 4. *Let $\lceil \cdot, l_o^i, l_f^i, \cdot \rceil, i = 1 \ldots n$ be the join points generated during (and including) the execution of $\mathring{R}$ [pointcut] and $\mathring{R}$ [advice] in absence of other aspects, for some l_o^i, l_f^i.*

If there is a restriction in the stack, $\forall$ applications of $\mathring{R}$, $\exists i, \langle \cdot, \cdot, \cdot, E[\texttt{jp}(\lceil \cdot, l_o^i, l_f^i, \cdot \rceil, \cdot)] \rangle \hookrightarrow \langle \cdot, \cdot, \cdot, E[\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\, W[\![\alpha$ where $\alpha \neq \epsilon$, then the computation ends with an error or diverges.

Proof. By analysis of the weaving process.

(Base case) We know that there is at least one aspect observing the join point $\lceil \cdot, l_o^i, l_f^i, \cdot \rceil$, then there are three possibilities:

- $(\cdot, \cdot, U) \in \alpha$
- $(\cdot, \cdot, \mathring{\text{R}}) \in \alpha$
- Both $(\cdot, \cdot, l_U) \in \alpha$ and $(\cdot, \cdot, \mathring{\text{R}}) \in \alpha$.

We can abstract from the presence of $\mathring{\text{R}}$ and/or U and only reason based on the position of U and $\mathring{\text{R}}$ in α. There are two cases:

(1) $U[pointcut]$ is evaluated first (because U appears after $\mathring{\text{R}}$ in α, or because $\mathring{\text{R}} \notin \alpha$):

(a) A $\texttt{jp}(\lceil \texttt{pc-exec}, U, \cdot, \cdot \rceil, \cdot)$ is generated (rule C/ASP), and reduced to $\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\, W[\![\alpha']\!]_{\cdot,\cdot}, \cdot))$ (rule WEAVE).

(b) Since $\mathring{\text{R}} \in \texttt{asps}(U)$, $\mathring{\text{R}} \in \alpha'$ (rule WEAVE).

(c) There are two cases:

i. Some U is after $\mathring{\text{R}}$ in α, therefore its $U[pointcut]$ is evaluated first. In this case, we are in the same situation as case (1.a).

ii. $\mathring{\text{R}}$ $[pointcut]$ is evaluated first. This corresponds to case (2).

(2) $\mathring{\text{R}}$ $[pointcut]$ is evaluated first (because $\mathring{\text{R}}$ appears after U in α, or because $U \notin \alpha$):

(a) The pointcut of $\mathring{\text{R}}$ potentially produces m join points. If at least one of these is observed, this leads us to the (base case).

(b) If none of these m join points is observed, there are three possibilities:

i. The underlying computation associated to these m join points or the computation between them end with an error.

ii. The pointcut of some $U \in \alpha$ is evaluated after the pointcut of $\mathring{\text{R}}$. In this case, we return to case (1.a).

iii. Only $\mathring{\text{R}}$ is in α (*i.e.* $\alpha = (\cdot, \cdot, \mathring{\text{R}})$). In this case, $\mathring{\text{R}}$ $[advice]$ is evaluated, which produces potentially produces $n - m$ join points. There are two cases again:

A. The underlying computation associated to the join points before the one that is observed or the computation between them end with an error.

B. The computation reaches the join point that is observed, which leads to the (base case) again.

As demonstrated by the previous case analysis, there is no other alternative for the evaluation to return indefinitely to the same cases if at least one of the join points generated during (and including) the execution of $\mathring{\text{R}}$ [pointcut] and $\mathring{\text{R}}$ [advice] is observed. The only escape is when an error is generated. Consequently, the computation in this scenario, ends with an error or diverges.

Note that, thanks to Lemma 3, the pointcuts of U are not evaluated. This is the reason why they never can produce an error (this is explicit in case 1.a) □

Corollary 1. *By negating the proposition in Lemma 4, we obtain:*

If a protected aspect is in the stack, and the computation does not end with an error or converges, then $\exists$ *an application of* $\mathring{R}$*,*
$\langle \cdot, \cdot, \cdot, E[\texttt{jp}(\lceil \cdot, l_o^i, l_f^i, \cdot \rceil, \cdot)] \rangle \hookrightarrow \langle \cdot, \cdot, \cdot, E[\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\, W[\![\alpha]\!]_{\cdot,\cdot,\cdot}, \cdot))] \rangle$ *where* $\alpha = \epsilon$*, for some* l_o^i, l_f^i*.*

Corollary 1 implies that there is only one way of not ending with an error or diverging once a protected aspect is in the stack and $\mathring{\text{R}}$ is participating: that for at least one application of $\mathring{\text{R}}$, the join points generated during (and including) the execution of $\mathring{\text{R}}$ [pointcut] and $\mathring{\text{R}}$ [advice] are not seen by any aspect.

When no aspects inhibit $\mathring{\text{R}}$, it is direct to prove that $\langle \mu, \alpha, J, E[\texttt{jp}\ \lceil \texttt{pc-exec}, U, \cdot, \cdot \rceil] \rangle \hookrightarrow^* \langle \mu', \alpha, J, E[v] \rangle$, for $v =$ **false** or $v = (\textbf{err}\ \cdot)$. We provide a test file that exercises this case (the file named uro.rkt). The desired result can be directly obtained by running the test with the executable semantics in PLT Redex. The error case can be obtained by producing an error during the evaluation associated to $\mathring{\text{R}}$. Both elements are available on-line.

Now that we know that $\mathring{\text{R}}$ (without considering errors) will eventually rule at least one U, we can straightforwardly conclude that a sequence of implicit interferences of the form: $\mathring{\text{R}}$ is observed by U, then this U is observed by $\mathring{\text{R}}$, the again (possible another) U observes $\mathring{\text{R}}$, and so on; will actually end with $\mathring{\text{R}}$. We call this kind of inhibition "vertical inhibition".

In a vertical inhibition, all pointcuts of all U will reduce to **false**. This is because the $\mathring{\text{R}}$ that preventively inhibits the first U, prevents this U from inhibiting the application of $\mathring{\text{R}}$ "below". The same will be true until $\mathring{\text{R}}$ or U are woven together (*i.e.* $\exists \lceil k, l_o, l_f, p \rceil$, for some k, l_o, l_f, p, so that $\langle \cdot, \cdot, \cdot, E[\texttt{jp}(\lceil k, l_o, l_f, p \rceil, \cdot)] \rangle \hookrightarrow \langle \cdot, \cdot, \cdot, E[\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\, W[\![\alpha]\!]_{\cdot,\cdot,\cdot}, \cdot))]$ where $\mathring{\text{R}}, U \in \alpha$. We call this situation "horizontal inhibition": U tries to inhibit $\mathring{\text{R}}$ when both are woven together.

Theorem 2 (Non-inhibition)**.** *If* l_{asp} *is a valid aspect reference in* μ*,* $\mathring{R} \in \texttt{asps}(l_{asp})$*, and* $\lceil \cdot, s, \cdot, \cdot \rceil \in J$*; where* $s \in PA$*, the set of protected aspects, then:*
If $\langle \mu, \alpha, J, E[\texttt{jp}(\lceil \texttt{pc-exec}, l_{asp}, \cdot, \cdot \rceil, \cdot)] \rangle \hookrightarrow^* \langle \cdot, \alpha, J, E[v] \rangle$*, then* $v =$ ***false*** *or* $v = (\textbf{\textit{err}}\ \cdot)$*.*

Proof. The key point in this demonstration is that vertical interference is resolved before horizontal interference. This is direct by the definition of the weaving function W: the weaving of a pointcut is completely evaluated before the weaving of the pointcut immediately after.

First, we know by the application of rule WEAVE that
$\langle \cdot, \alpha, J, E[\texttt{jp}\ \lceil \texttt{pc-exec}, l_{asp}, \cdot, \cdot \rceil] \rangle \hookrightarrow$
$\langle \cdot, \alpha, \lceil \texttt{pc-exec}, l_{asp}, \cdot, \cdot \rceil + J, E[\texttt{in-jp}(\texttt{swap}(\texttt{app/prim}\ W[\![\alpha]\!]_{\cdot,\cdot,\cdot}))] \rangle$, where $\mathring{\text{R}}$,$U \in \alpha$. Then:

1. Thanks to Corollary 1, we know that the pointcuts of all U evaluate to **false**, **err** $\cdot$, or diverge. This holds no matter the position of U in α. This is because vertical inhibition es prevented first.
2. The pointcut of $\mathring{\text{R}}$ evaluates to **true**, and since no other pointcut of an aspect in α evaluates to **true**, the advice of $\mathring{\text{R}}$ forces the the pointcut of U to reduce to **false**. (If an non-trapped error is produced during the evaluation of $\mathring{\text{R}}$, then the computation ends with an error).

The use of `swap` and `in-jp` ensures that the original α and J are restored. The second application of `swap` (in the base case of the W metafunction) restores α. The application of `in-jp` restores the original J, previously changed by rule WEAVE. □