CHAPTER 8

Trace-based Aspects

Remi Douence, Pascal Fradet, and Mario Südholt

This chapter presents *trace-based aspects*, a model which takes into account the history of program execution in deciding what aspect behavior to invoke. That is, aspect behavior is invoked depending on relations among events of the execution history. With trace-based aspects, weaving is accomplished through an execution monitor which modifies the program execution to include the behavior specified by the aspects. We motivate trace-based aspects and explore the trade-off between expressiveness and property enforcement/analysis.

More concretely, we first present an expressive model of trace-based aspects enabling proofs of aspect properties by equational reasoning. Using a restriction of the aspect language to regular expressions, we address the difficult problem of interactions between conflicting aspects. Finally, by restricting the actions performed by aspects, we illustrate how to keep the semantic impact of aspects under control and to implement weaving statically.

8.1. INTRODUCTION

Aspect-Oriented Programming (AOP) is concerned with providing programmatic means to modularize crosscutting functionalities of complex applications. By encapsulating such functionalities into aspects, AOP facilitates development, understanding and maintenance of programs. An important characteristics of aspects is that they are built from *crosscuts* (pointcuts), which define where an aspect modifies an application, and *inserts* (advice), which define the modifications to be applied. Typically, crosscuts denote sets of program points or execution points of the base application and inserts are expressed in a traditional programming language. For instance, an aspect for access control could be defined in terms of crosscuts denoting sets of access methods and inserts performing the access control tests. However, because aspect languages are rather limited, it is often necessary to use inserts to pass information from one crosscut to another one. Consider, for example, an aspect performing some access control for logged-in users. An aspect language that cannot refer to the historical login event must use an extra insert to set a flag in the login code, and must write the access control aspect to recognize this flag.

Conventional aspect languages allow unrestricted inserts and the role of inserts overlaps with that of crosscuts. This makes reasoning on aspects and woven programs difficult. In this chapter, we present an approach which—by means of expressive aspect languages and restrictions on inserts — enables reasoning about different aspect properties.

Trace-Based Aspects are defined on traces of events occurring during program execution. Trace-based aspects are more expressive than those based on atomic points because they can express relations between execution events, including those involving information from the corresponding execution states. For example, an aspect for access control could express that a user has to log in first in order to pass an access check later. Such aspects are called *stateful*: their implementation must use some kind of state to represent their evolution according to the event encountered. Conceptually, weaving is modeled by an execution monitor whose state evolves according the history of program execution and which, in case of a match, triggers the execution of the corresponding action. By strictly separating crosscuts and inserts by means of two different, well-defined languages, we address the formalization of aspects and weaving. Restrictions on these languages allow us to design static analysis of aspect properties as well as an optimized implementation of aspect weaving.

In Section 0, we informally introduce the main features of trace-based AOP: observable execution traces, stateful aspects (composed of crosscuts and inserts), and weaving (based on execution monitoring). In Sections 0-0, we explore three different options within the trade-off between expressiveness and property enforcement/analysis. The first provides an expressive crosscut language with no restrictions on the inserts. However, the expressive power of this option precludes automatic proofs of most aspect properties. The second option is characterized by more restricted, but still stateful, aspects corresponding to regular expressions over execution traces. Because of this restriction it is possible to statically detect whether several aspects interact (e.g., testing whether an encryption aspect interacts with a system logging aspect). We also suggest operators for the resolution of such interactions. The last option is characterized by a very restricted insert language where aspects can be seen as formal safety properties. We present how these aspects/properties can be statically and efficiently woven. An application of this technique is the securization of mobile code upon receipt. Finally, we discuss related work and conclude in Section 0.

This chapter is a unified presentation of three distinct studies [3], [7] (which has recently been extended in [8]), and [9] sharing a trace-based approach to AOP. In order to make the presentation more intuitive, we have deliberately omitted many extensions and technical details.

8.2. Characteristics of Trace-Based Aspects

Trace-based aspects have two main characteristics. First, aspects are defined over sequences of observable execution states. Second, weaving is more naturally performed on executions rather than program code. The weaver can be seen as a monitor interleaving the execution of the base program and execution of inserts.[1]

8.2.1. Observable Execution Trace

The base program execution is modeled by a sequence of observable execution states (a.k.a. join points). This trace can be formally defined on the basis of a small-step semantics [16] of the programming language. Each join point is an abstraction of the execution state. Join points may denote not only syntactic information (e.g., instructions) but also semantic information (e.g., dynamic values). For example, when the user Bob logs in, the function `login()`

[1] Note that this model does not preclude implementing weaving as a compile-time process (see Section 0) .

is called in the base program with `"Bob"` as a parameter. This join point of the execution can be represented by the term `login("Bob")`.

8.2.2. Aspect Language

The basic form of an aspect is a rule of the form $C \blacktriangleright I$ where C is a crosscut and I is an insert. The insert I is executed whenever the crosscut C matches the current join point. Basic aspects can be combined using operators (sequence, repetition, choice, etc.) to form stateful aspects.

Crosscuts. A crosscut defines execution points where an aspect should perform an action. In general, a crosscut C is a function that takes a join point as a parameter. This function returns either `fail` when the join point does not satisfy the crosscut definition, or a substitution that captures values of the join point. For example, we can define a crosscut *isLogin* that matches session logins and captures the corresponding user name. It would return `fail` when it is applied to the join point `logout()` and the substitution *uid* = `"Bob"` when it is applied to `login("Bob")`.

Inserts. An insert is an executable program fragment with free variables. For instance, the insert `addLog(`*uid*` + "logged in")` prints the name of a logged user when it is executed. In this insert, the name of the user is represented by the variable *uid* to be bound by a crosscut. In the remainder of the paper, the special insert `skip` represents an instruction doing nothing.

Stateful aspects. The intuition behind a basic aspect *C* ► *I* is that when *C* matches the current join point and yields a substitution ϕ, the program ϕ *I* is executed. For example, we can define a basic security aspect which logs sessions as follows:

isLogin ► `addLog(`*uid*` + "logged in")`

In order to build stateful aspects, basic aspects can be combined using control operators. Using a C-like syntax, we can define an aspect which logs all sessions as follows:

`while(true){` *isLogin* ► `addLog(`*uid*` + "logged in") }`

This definition applies the basic security aspect again and again. Control operators allow us to define sophisticated aspects on execution traces. For instance, the following aspect tracks sequences of sessions (`login` followed by `logout`).

```
while(true){ isLogin ▶ addLog (uid + "logged in") ;
             isLogout ▶ addLog(uid + "logged out") }
```

8.2.3. Weaving

In general, several aspects addressing different issues (e.g., debugging and profiling) can be composed (using a parallel operator ||) and woven together. The weaver takes a parallel composition of *n* aspects A_1 || … || A_n and tries to apply each of them (in no specific order) at each join point of the execution trace.

Conceptually, the weaver is an execution monitor that selects the current basic aspects of A_1 , … , A_n and tries to apply them at each join point. When a crosscut matches the current join point, the corresponding insert is executed. After all basic aspects have been considered, the base program execution is resumed and proceeds until the next join point.

When a basic aspect of a stateful aspect A_i has been applied and its insert executed, the state of A_i evolves. The control structure of A_i (e.g., repetition or sequence) specifies which basic aspect must be considered next. For instance, the previous security aspect remains in its initial state until a login occurs. After the aspect has matched a `login` event, it waits to match a `logout` event before returning to its initial state.

In the remainder of this chapter, we instantiate this framework to form a variety of different definitions of crosscuts, inserts and stateful aspects. We thus obtain different aspect languages and enable reasoning about aspect-oriented programs, bothmanually and, using static analysis techniques, automatically.

8.3. EXPRESSIVE ASPECTS AND EQUATIONAL REASONING

We present a first instantiation of the general framework for AOP introduced in the previous section. This instantiation, which is inspired by the work presented in [9], is intended to illustrate two main points:

- The usefulness of expressive aspect definitions.
- The application of general proof techniques for the analysis and transformation of AO programs.

Crosscuts. In this section we instantiate the general framework by allowing crosscuts *C* to be arbitrary predicates. For instance, a predicate *isWeakPassword* could recognize the event of changing a password to a word a dictionary. Note that we do not define the language of crosscuts; they are to be defined using some general-purpose language.

Stateful Aspects. Aspects are defined as a collection of mutually recursive definitions of the form *var* = *A*. Since one of our main interests lies in the definition of *stateful* crosscuts, we base an aspect definition on the following grammar:

$A ::= C \blacktriangleright I$	; basic aspect
$\mid A_1 ; A_2$	; sequence
$\mid A_1 \blacksquare A_2$	; choice
$\mid var$	; invocation

The grammar allows us to compose aspects by sequentialization, deterministic choice and aspect invocation. In a deterministic choice $A_1 \blacksquare A_2$, A_1 is always chosen if it is applicable; A_2 is chosen only if it is applicable and A_1 is not. Using composed aspects, we can define, for example, an aspect *tryOnce* trying to apply $C \blacktriangleright I$ only on the current join point and doing nothing afterward as

$$tryOnce = (C \blacktriangleright I ; void) \blacksquare void$$

$$void = isAny \blacktriangleright \texttt{skip} ; void$$

If *C* matches the current join point, the weaver chooses the first branch, executes the insert *I* and the aspect becomes *void* that keeps doing nothing. Otherwise, the weaver chooses the second branch (*void*) which keeps doing nothing right from the start.

In order to illustrate how such expressive aspects may be used, consider the following definition:

$$
\begin{aligned}
log &= (isLogin \blacktriangleright \texttt{addLog(}uid\texttt{)};\, log\,;\, isLogout \blacktriangleright \texttt{skip)} \\
&\qquad \blacksquare\ isLogout \blacktriangleright \texttt{skip} \\
logNestedLogin &= isLogin \blacktriangleright \texttt{skip}\ ;\, log\,;\, logNestedLogin
\end{aligned}
$$

The aspect *logNestedLogin* considers sessions starting with a call to the `login` function with the user identifier as a parameter (whose occurrence is matched by the crosscut *isLogin*) and ending with a call to the function `log-out` (matched by the crosscut *isLogout*). This aspect logs nested (i.e. non top-level) calls to the login function, which may be useful because such a call may login into a non-local network and be therefore dangerous. The recursive definition of the aspect is responsible for pairing logins and logouts, thus detecting non top-level calls to `login`.

Now, let us consider the following aspect:

initAtFirstLogin = *isLogin* ► `initNetworkInfo()` ; *void*

This aspect *initAtFirstLogin* detects the first call to login in order to initialize network information. Then the following calls to login are ignored.

It is easy to prove that the two aspects *logNestedLogin* and *initAtFirst-Login* are equivalent to a single sequential aspect. Basically, this can be proven by unfolding of recursive definitions and induction principles [9]. The proof starts with a parallel composition *logNestedLogin* || *initAtFirst-Login* and eliminates the parallel operator by producing all the possible pairs of crosscuts from the two aspect definitions and by folding. The resulting sequential aspect can be simplified if a pair of crosscuts has no solution. In our example we get the following sequential aspect:

initAndLog = *isLogin* ► `initNetworkInfo()`; *log* ; *logNestedLogin*

8.4. DETECTION AND RESOLUTION OF ASPECT INTERACTIONS

By restricting the expressiveness of our aspect language (while still adhering to stateful aspects), it is possible to automatically prove (certain) aspect properties. In this section we consider a second instantiation of the general framework that supports a more restrictive yet expressive crosscut language in which static checking of interactions is feasible.

Crosscuts. A crosscut is defined by conjunctions, disjunctions and negations of terms:

$$C \;::=\; T \;\mid\; C_1 \;\mathbf{and}\; C_2 \;\mid\; C_1 \;\mathbf{or}\; C_2 \;\mid\; \mathbf{not}\; C$$

where *T* denotes terms with variables. The formulas used to express these crosscuts belong to the so-called quantifier-free equational formulas [5]. Whether such a formula has a solution is decidable. This is one of the key properties making the analysis in this section feasible.

We can define, for example, a crosscut matching logins performed by the user `root` on any machine, or by any non-root user on any machine but the `server` as follows:

$$\texttt{login(root,}\,m\texttt{)}\ \textbf{or}\ \texttt{(login(}u,m\texttt{)}\ \ \textbf{and}\ \texttt{(}\textbf{not}\ \texttt{login(}u,\texttt{server)))}$$

In this context, checking whether the current join point (which, remember, is represented by a term) matches the crosscut definition is computed by a generalized version of the unification algorithm, that is well-known, e.g., from logic programming.

Note that, for the sake of decidability (i.e. static analyses) the crosscuts C defined by the equation above are less expressive than those considered in the previous section. They can only denote join points as term patterns (as opposed to arbitrary term predicates).

Stateful aspects. The main idea of the aspect language presented in this section is to restrict stateful aspects to regular expressions using the following grammar:

$A ::= C \blacktriangleright I \,;\, A$	; sequence
$\mid\ C \blacktriangleright I \,;\, var$	; end of sequence
$\mid\ A_1 \blacksquare A_2$	; choice

Using this aspect language a security aspect *logAccess* that logs file accesses (calls to `read`) from a call to `login` until a call to `logout` (assuming non-nested sessions) can be expressed as:

$$logAccess = \texttt{login}(u, m) \blacktriangleright \texttt{skip} \,;\, logRead$$

$$logRead = (\texttt{logout()} \blacktriangleright \texttt{skip} \,;\, logAccess)$$

$$\blacksquare\, (\texttt{read}(x) \blacktriangleright \texttt{addLog}(x) \,;\, logRead)$$

where x matches the name of the accessed file.

8.4.1. Aspect Interactions

Remember that a parallel composition of n aspects $A_1 \,||\, \ldots \,||\, A_n$ does not define any specific order of application of aspects; the result of weaving may be non-deterministic. This situation arises when aspects interact, that is to say when at least two inserts must be executed at the same join point. For instance, consider the following aspect:

cryptRead = `read(`x`)` ► `crypt(`x`)`; *cryptRead*

This aspect states that the reads should be encrypted. It obviously interacts with the aspect *logAccess* defined above which describes logging for all users. When a user logs in and accesses a file, this access must be logged *and* the file name must be encrypted.

The algorithm to check aspects interaction is similar to the algorithm for finite-state product automata. It terminates due to the finite-state nature of our aspects. Starting with a composition $A \parallel A'$, the algorithm eliminates the parallel operator by producing all the possible pairs (conjunction) of crosscuts from A and A'. A conjunction of crosscuts C_1 **and** C_2 is a solvable formula and we can check if it has a solution using the algorithm of [5]. A crosscut with no solution cannot match any join point and can be removed from the aspect [7]. In the case of the example *logAccess* || *cryptRead*, we get:

$$\begin{aligned}
&\mathit{logAccess} \parallel \mathit{cryptRead} \\
&\quad = \mathtt{login}(u, m) \blacktriangleright \mathtt{skip}\ ; \mathit{logCrypt} \\
&\quad \blacksquare\ \mathtt{read}(x) \blacktriangleright \mathtt{crypt}(x)\ ;\ \mathit{logAccess} \parallel \mathit{cryptRead} \\
&\mathit{logCrypt} = \mathtt{logout}() \blacktriangleright \mathtt{skip}\ ;\ \mathit{logAccess} \parallel \mathit{cryptRead} \\
&\quad \blacksquare\ \mathtt{read}(x) \blacktriangleright (\mathtt{addLog}(x) \blacklozenge \mathtt{crypt}(x))\ ;\ \mathit{logCrypt}
\end{aligned}$$

Conflicts are represented using the non-deterministic function ($I_1 \blacklozenge I_2$) which returns either $I_1\ ;\ I_2$ or $I_2\ ;\ I_1$. Here, we have ($\mathtt{addLog}(x) \blacklozenge \mathtt{crypt}(x)$), so the two aspects are not independent. Note that spurious conflicts have already been eliminated with the help of the rule

$$(I \blacklozenge \mathtt{skip}) = (\mathtt{skip} \blacklozenge I) = I.$$

This analysis does not depend on the base program to be woven. When there is no ($\blacklozenge$) in the resulting sequential aspect, the two aspects are independent for all programs. This property does not have to be checked again after any program modification. However, this property is a sufficient but not a necessary condition for aspect interaction. A more precise analysis is possible by taking into account the possible sequences of join points generated by the base program to be woven [7].

8.4.2. Support for Conflict Resolution

When no conflicts have been detected, the parallel composition of aspects can be woven without modification. Otherwise, the programmer must get rid of the nondeterminism by making the composition more precise. In the following, we present some linguistic support aimed at resolving interactions.

The occurrences of rules of the form $C \blacktriangleright (I_1 \blacklozenge I_2)$ indicate all potential interactions. They can be resolved one by one. For each $C \blacktriangleright (I_1 \blacklozenge I_2)$, the programmer may replace each rule $C \blacktriangleright (I_1 \blacklozenge I_2)$ by $C \blacktriangleright I_3$ where I_3 is a new insert which combines I_1 and I_2 in some way. For instance, in the previous example, (`addLog(x)` ♦ `crypt(x)`) can be replaced by `crypt(x) ; addLog(x)` in order to generate encrypted logs.

This option is flexible but can be tedious. Instead of writing a new insert for each conflict, the programmer may indicate how to compose inserts at the aspect level. We propose a parallel operator $||_{seq}$ to indicate that whenever a conflict occurs, (I_1 ♦ I_2) must be replaced by I_1 ; I_2 (where “;” denotes the sequencing operator of the programming language). Other parallel operators are useful, such as $||_{fst}$ which replaces (I_1 ♦ I_2) by I_1 only.

Let us reconsider the two aspects *logAccess* and *cryptRead*:

- *logAccess* $||_{seq}$ *cryptRead* generates plaintext logs for super users.
- *cryptRead* $||_{seq}$ *logAccess* generates logs for users by logging (possibly encrypted) accesses.

8.5. Static weaving of safety properties

The previous restrictions allow detecting interactions during weaving. However, they are not sufficient to detect semantic interactions. The code inserted by an aspect may still influence the application of another independent aspect. Our notion of independence only ensures that aspects can be woven in any order. In order to prevent semantic interactions and, more generally, to control the semantic impact of weaving, one has to restrict the language of inserts. Here, we consider the same aspect language as the previous section, except for the language of inserts which becomes

$$I \ ::= \ \texttt{skip} \mid \texttt{abort}$$

Even if this restriction is quite drastic (aspects can only abort the execution), interesting aspects can still be expressed. The expressive crosscut language allows us to specify safety properties (properties stating that no "bad thing" happens during the execution). Aspects can be used to rule out unwanted execution traces and to express security policies [3].

This restriction has several benefits:

- Aspects are semantic properties and the impact of weaving is clear.
- Inserts always commute. There are no interactions between aspects which can be composed freely in parallel.

The woven program satisfies the property/aspect: for executions in accordance with the property, it has the same behavior as the base program. Otherwise, it produces an exception and terminates just before violating the property.

The main drawback of execution monitors is their runtime cost. They are not specialized to the program and each program instruction may involve a runtime check. In the remainder of this section we present how to weave such trace-based aspects statically and efficiently.

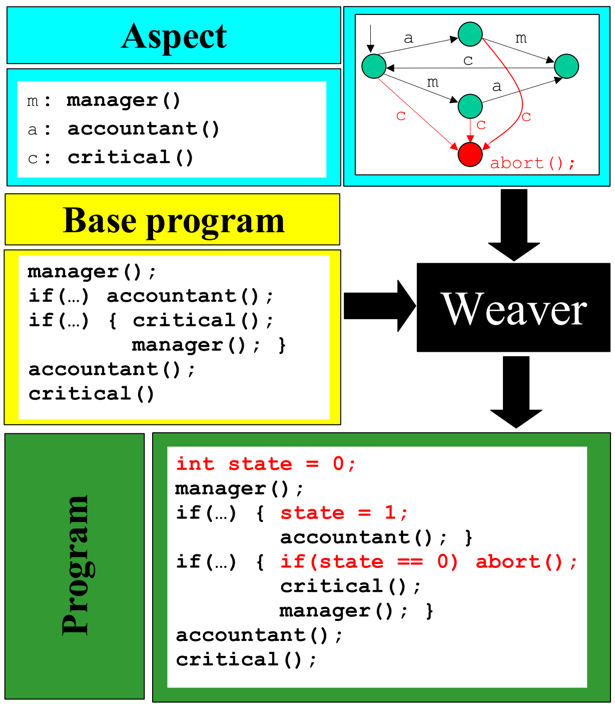
8.5.1. Example

Consider the following aspect

$$\begin{aligned}
\mathit{safe} = {} & \texttt{accountant()} \blacktriangleright \texttt{skip}\ ;\\
& \quad (\texttt{manager()} \blacktriangleright \texttt{skip}\ ; \texttt{critical()} \blacktriangleright \texttt{skip}\ ; \mathit{safe}\\
& \quad \blacksquare\, \texttt{critical()} \blacktriangleright \texttt{abort}\ ; \mathit{safe})\\
& \blacksquare\, \texttt{manager()} \blacktriangleright \texttt{skip}\ ;\\
& \quad (\texttt{accountant()} \blacktriangleright \texttt{skip}\ ; \texttt{critical()} \blacktriangleright \texttt{skip}\ ; \mathit{safe}\\
& \quad \blacksquare\, \texttt{critical()} \blacktriangleright \texttt{abort}\ ; \mathit{safe})\\
& \blacksquare\, \texttt{critical()} \blacktriangleright \texttt{abort}\ ; \mathit{safe}
\end{aligned}$$

The property defined by the aspect states that a critical action cannot take place before the clearance of the manager and the accountant (i.e. at least a call to `manager` and to `accountant` must occur before each call to `critical`).

Figure 8-1 illustrates weaving of this property on a very simple imperative base program.



```
m: manager()
a: accountant()
c: critical()
```

```
manager();
if(…) accountant();
if(…) { critical();
        manager(); }
accountant();
critical()
```

```
int state = 0;
manager();
if(…) { state = 1;
        accountant(); }
if(…) { if(state == 0) abort();
        critical();
        manager(); }
accountant();
critical();
```

Figure 8-1 Weaving *safe* on a simple imperative base program.

Since the property is specified by a finite state aspect, it can be encoded as an automaton with alphabet `{m, a, c}` corresponding to the calls to `manager`, `accountant` and `critical` respectively. Notice that the base program may violate this property whenever the condition of the first `if` statement is false. The woven program, where two assignments and a conditional have been inserted, satisfies the property (i.e. aborts whenever the property is about to be violated).

An important challenge is to make this dynamic enforcement as inexpensive as possible. In particular, if we are able to detect statically that the base program satisfies the property, then no transformation should be performed.

8.5.2. Weaving phases

Our aspects define a regular set of allowed finite executions. An aspect is encoded as a finite state automaton over events. The language recognized by the automaton is the set of all authorized sequences of events.

The weaver is a completely automatic tool which takes the automaton, the base program and produces an instrumented program [3]. We now outline its different phases (depicted in Figure 8-2).

Base Program Annotation. The first phase is to locate and annotate the instructions of the base program corresponding to events (crosscuts). Depending on the property we want to enforce, the events can be calls to specific methods, assignments to specific variables, opening of files, etc. A key constraint is that an instruction of the base program must be associated with at most one event. This is easy to ensure when events are specified solely based on the syntax. In order to take semantic crosscuts into account, the base program must be transformed beforehand. Consider the event "*x is assigned the value 0*", it cannot be statically decided whether an assignment `x:=e` will generate this event or not. A solution is to transform each assignment `x:=e` into the statement `if e=0 then x:=e else x:=e` where each instruction is now associated with a single event. Such pre-transformations rely on static program analyses to avoid insertion of useless tests.

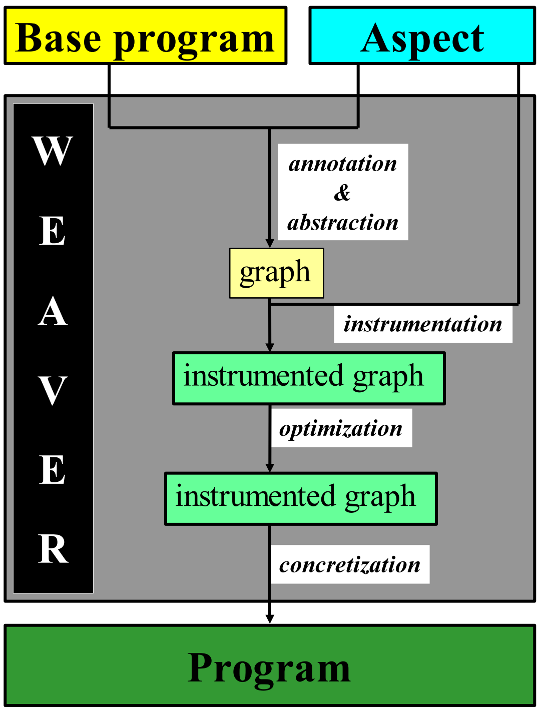


Figure 8-2 Phases of weaving.

Base Program Abstraction. The base program is abstracted into a graph whose nodes denote program points and edges represent instructions (events). The abstraction makes the next two phases independent of a specific programming language. Since the aspect is a trace property, the abstraction is the control-flow graph of the base program. In order to produce a precise abstraction, this phase relies on a control-flow analysis.

Instrumentation. The next phase is to transform the graph in order to rule out the forbidden sequences of events. We integrate the automaton by instrumenting the graph with additional structures (states and transition functions) that mimic the evolution of the automaton. Intuitively, this instrumentation corresponds to the insertion of an assignment (to implement the state transition of the underlying automaton) and a test (to check whether the property is about to be violated) before each event. This naive weaving is optimized by the next phase.

Optimizations. The instrumented graph is refined in three steps. First, the automaton specifies a general property independent of any particular program. The first step is to *specialize* the automaton with respect to the base program. Second, the second step yields a normalized instrumented graph using a transformation similar to the classical automaton *minimization*. Finally, the last optimization removes useless state transitions using static analyses.

The graph after optimization represents a program where at most one test and/or assignment (state transition) have been inserted at each `if` and `while` statement.

Concretization. The optimized graph must be translated back into a program. The graph has remained close to the base program since its nodes and edges still represent the same program points and instructions. We just need a way to store, fetch, and test a value (the automaton state) without affecting the normal execution. This is easily done using a fresh global variable.

8.5.3. Just-in-time Weaving

The most interesting application of this technique is securing mobile code on receipt. The local security policy is declared as a property (aspect) to be enforced on incoming applets. The just-in-time weaver secures (i.e. abstracts, instruments, optimizes, and concretizes) an applet before loading it. Since some steps are potentially costly, our implementation uses simple heuristics that make the time complexity of weaving linear in the size of the program.

There are several benefits to this separation of security concerns. First, it is easier to express the policy declaratively as a property. Second, the approach is flexible and can accommodate customized properties. This feature is especially important in a security context where it is impossible to foresee all possible attacks and where policies may have to be modified quickly to respond to new threats.

8.6. CONCLUSION

In this chapter, we have presented a model (and three instantiations) for AOP based on execution traces. We have focused on the following points:

- Expressive and stateful aspect definitions.
- A model conceptually based on weaving of executions.
- Reasoning about and analysis of aspect properties, in particular aspect interaction.
- Enforcement of properties by program transformation (i.e. static weaving).

We now briefly consider these contributions in turn and compare our approach with related work.

We have advocated and presented expressive (i.e. stateful) aspect languages. The crosscut language of AspectJ [14] consists mostly in *single instruction patterns* matching events such as a method calls or field accesses. AspectJ's patterns are very similar to our basic aspects $C \blacktriangleright I$. Expressing stateful aspects in AspectJ requires bookkeeping code in advice to pass information between crosscuts (e.g., increment a counter in an advice to check for the counter value later). The AspectJ construction `cflow` is the only exception allowing the definition of a form of stateful aspect. For example, `cflow(call(critical)) && call(read)` matches join points where `read` is called whenever there is a pending call to `critical` in the execution stack.

Our model is conceptually based on a monitor observing execution events and weaving execution traces, similar to the approach proposed by Filman and Havelund [11]. Other techniques can be related to execution monitors. Computational reflection is a general technique used to modify the execution mechanisms of a programming language. Restricted approaches to reflection have been proposed in order to support AOP. For instance, the composition filter model [2] proposes method wrappers in order to filter method calls and returns. De Volder *et al.* [6] propose a meta-programming framework based on Prolog. Unfortunately, these approaches do not allow stateful aspects.

By appropriate restrictions of the aspect language we have proposed some solutions to the difficult problem of aspect interactions. Few other program analyses for AO programs have been proposed, most notably by Sereni and de Moor for the optimization of AspectJ's `cflow` construction [18] and by Shiman and Katz to verify applicability of aspects using model checking techniques [19]. At the tool-level, ASPECTJ provides limited support for aspect interaction analysis using IDE integration: the base program is annotated with crosscutting aspects. This graphical information can be used to detect conflicting aspects. However, the simple (i.e. stateless) crosscut model of ASPECTJ would entail an analysis detecting numerous spurious conflicts because the bookkeeping code cannot be taken into account. In case of real conflicts, ASPECTJ programmers must resolve conflicts by reordering aspects using the keyword `dominate`. When two aspects are unrelated *w.r.t.* the domination or hierarchy relations, the ordering of inserts is undefined.

In order to define static interaction analysis we had to formally define aspects and weaving (see [7] for a formal treatment of Section 4). There are several approaches to the formalization of AOP. Wand *et al.* [22] propose a denotational semantics for a subset of AspectJ. Lämmel [15] formalizes method call interception with big-step semantics. Andrews' model [1] relies on algebraic processes. He focuses on equivalence of processes and correctness (termination) of the weaving algorithm. Other operational approaches to the formalization of AO systems have been proposed by Walker *et al.* [21] and Tucker and Krishnamurthi [20] using different forms of abstract machines, as well as Jagadeesan *et al.* [13] based on process calculi.

Finally, we have shown in Section 0 that by restricting the insert language, aspects can be seen as formal properties which can be enforced by program transformation. Dynamic monitors (such as VeriSoft [12] and Amos [4]) or "security kernels" (such as Schneider's security automata [17]) have been used to enforce security properties. By contrast, our programming language approach permits many optimizations and avoids extending the runtime system or the language semantics.

The different aspect languages presented suggest several extensions. For example, allowing crosscuts of the same aspect to share variables makes the aspect language more expressive (cf. [8]). The possibility of associating an instance of an aspect with a run-time entity (e.g. each instance of a class in a Java program) would facilitate the application of our model to object-oriented languages. It would been interesting to characterize a larger class of inserts (beyond `abort`) allowing to keep the semantic impact of weaving under strict control. More generally, we believe that an important avenue for further AOP research is to provide more safeguards in terms of static analyses and specially-tailored aspect languages. In order to make these analyses applicable in the context of large-scale application, their integration with standard model checking techniques [10] seems promising.

REFERENCES

1. ANDREWS, J. H. 2001. Process-algebraic foundations of aspect-oriented programming. In *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001),* (Kyoto), A. Yonezawa and S. Matsuoka, Eds. LNCS, vol. 2192. Springer-Verlag, Berlin, 187–209.

2. Bergmans, L. and Akşit, M. 2001. Composing crosscutting concerns using composition filters. *Comm. ACM 44*, 10 (Oct.), 51–57.

3. Cohen, D., Feather, M. S., Narayanaswamy, K., and Fickas, S. S. 1997. Automatic monitoring of software requirements. In *19th Int'l Conf. Software Engineering (ICSE),* (Boston). ACM, 602–603.

4. Colcombet, T. and Fradet, P. 2000. Enforcing trace properties by program transformation. In *27th Symp. Principles of Programming Languages (POPL),* (Boston). ACM, 54–66.

5. Comon, H. 1991. Disunification: A survey. In *Computational Logic: Essays in Honor of Alan Robinson,* J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, Massachusetts, 322–359.

6. De Volder, K. and D'Hondt, T. 1999. Aspect-oriented logic meta programming. In *Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection,* P. Cointe, Ed. LNCS, vol. 1616. Springer Verlag, Berlin, 250–272.

7. Douence, R., Fradet, P., and Südholt, M. 2002. A framework for the detection and resolution of aspect interactions. In *1st ACM Conf. Generative Programming and Component Engineering* (Pittsburgh). Springer-Verlag, Berlin, 173–188.

8. Douence, R., Fradet, P., and Südholt, M. 2004. Composition, reuse and interaction analysis of stateful aspects. In *Aspect-Oriented Software Development (AOSD),* (Lancaster). ACM, 141-150.

9. DOUENCE, R., MOTELET, O., AND SÜDHOLT, M. 2001. A formal definition of crosscuts. In *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001),* (Kyoto), A. Yonezawa and S. Matsuoka, Eds. LNCS, vol. 2192. Springer-Verlag, Berlin, 170–186.

10. FELTY, A. P. AND NAMJOSHI, K.S. 2003. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology 12,* 1 , 3–27.

11. FILMAN, R. E. AND HAVELUND, K. 2002.Realizing aspects by transforming for events. In *Automated Software Engineering (ASE),* IEEE.

12. GODEFROID, P. 1997. Model checking for programming languages using Verisoft. In *24th Symp. Principles of Programming Languages (POPL),* (Paris). ACM, 174–186.

13. JAGADEESAN, R., JEFFREY, A., AND RIELY, J. 2003. A calculus of untyped aspect-oriented programs. In *ECOOP 2003— European Conference on Object-Oriented Programming,* (Darmstadt), L. Cardelli, Ed. LNCS, vol. 2743. Springer-Verlag, Berlin, 54-73.

14. KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming, 15th European Conference,* (Budapest), J. L. Knudsen, Ed. LNCS, vol. 2072. Springer-Verlag, Berlin, 327–353.

15. LÄMMEL, R. 2002. A semantical approach to method-call interception. In *1st Int'l Conf. Aspect-Oriented Software Development (AOSD),* (Enschede, The Netherlands), G. Kiczales, Ed. ACM, 41–55.

16. NIELSON, F. AND NIELSON, H. R. 1992. *Semantics with Applications — A Formal Introduction*. Wiley, New York.

17. SCHNEIDER, F. B. 2000. Enforceable security policies. *ACM Transactions on Information and System Security 3*, 1 (Feb.), 30–50.

18. SERENI, D. AND DE MOOR, O. 2003. Static analysis of aspects. In *Aspect-Oriented Software Development (AOSD),* (Boston). ACM, 30-39.

19. Shiman, M. and Katz, S. 2003. Superimpositions and Aspect-Oriented Programming. The Computer Journal, *46*, 5 , 529-541

20. TUCKER, D.B. AND KRISHNAMURTI. 2003. Pointcuts and advice in higher-order languages. In *Aspect-Oriented Software Development (AOSD),* (Boston). ACM, 158-167.

21. WALKER, D., ZDANCEWIC, S., AND LIGATTI, J. 2003. A theory of aspects. In *Functional Programming,* (Upsala). ACM.

22. WAND, M., KICZALES, G., AND DUTCHYN, C. in press. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*.