Composition, reuse and interaction analysis of stateful aspects

Rémi Douence, Pascal Fradet, Mario Südholt

* *

Remi.Douence@emn.fr Pascal.Fradet@inria.fr Mario.Sudholt@emn.fr

Aspect-Oriented Programming promises separation of concerns at the implementation level. However, aspects are not always orthogonal and aspect interaction is a fundamental problem. In this paper, we extend previous work on a generic framework for the formal definition and interaction analysis of stateful aspects. We propose three important extensions which enhance expressivity while preserving static analyzability of interactions. First, we provide support for variables in aspects in order to share information between different execution points. This allows the definition of more precise aspects and to avoid detection of spurious conflicts. Second, we introduce generic composition operators for aspects. This enables us to provide expressive support for the resolution of conflicts among interacting aspects. Finally, we offer a means to define applicability conditions for aspects. This makes interaction analysis more precise and paves the way for reuse of aspects by making explicit requirements on contexts in which aspects must be used.

Keywords: aspect oriented programming, formal model, static analysis, aspect interactions, aspect composition, reuse of aspects.

Aspect-Oriented Programming (AOP) [10] promises the systematic treatment of separation of concerns at the implementation level. Research on AOP is far from being mature in many respects and there remain fundamental problems. In this paper we consider three of these problems: devising an appropriate notion of aspect composition, support for reuse of aspects, and automatic analysis of conflicts among non-orthogonal aspects. Currently, there is only a small body of work addressing such issues and even fewer such work with a sound formal basis. Most frequently, programmers dispose of only rudimentary notions of aspect composition and no explicit support for aspect reuse. Moreover, they are responsible for identifying interactions between conflicting aspects and have to implement conflict resolution code without support for this task.

*Partially funded by the EU project EASYCOMP (see www.easycomp.org), no. IST-1999-014191.

We address these problems based on the generic and formal framework introduced in [4]. This framework is very general: it does not depend on a specific programming language and is expressive enough to allow the definition of stateful aspects. *Stateful aspects* are defined in terms of sequences of join points; they take into account the history of computation instead of a single join point. Another major property of the framework is to permit the static and automatic analysis of interactions between (stateful) aspects.

In this article, we propose three important extensions to that framework. First, we augment the underlying aspect language by introducing variables allowing the sharing of information between different parts of an aspect. The language becomes much more expressive but the absence of interactions between aspects can still be checked statically. Second, we introduce new composition operators for aspects. Once again, this extension fits nicely within the framework and the static analyzability of interactions is preserved. These operators are particularly useful to resolve conflicts between interacting aspects. Interactions arise when distinct aspects match the same join points. Making the composition of aspects at such interaction points precise permits to resolve conflicts. Finally, we introduce a notion of explicit requirements on base programs for the applicability of aspects. We show how such contextual aspects can be used to make our interaction analysis more precise and how they support a notion of aspect reuse.

The paper is organized as follows. In Section 2, we introduce our formal framework and provide an informal overview of the extensions presented in the following three sections. Section 3 makes the underlying aspect language more expressive through support for inter-crosscut variables and presents an associated interaction analysis. Section 4 defines the new composition facilities and their application to conflict resolution. Section 5 introduces explicit requirements for aspects, discusses how this information makes interaction analysis more precise, and how it facilitates reuse of aspects. Finally, Section 6 presents related work and concludes.

In this section, we briefly present the generic framework (introduced by the authors in [4]) on which the current work is based. Then, we give an overview of the extensions defined in the current article for the analysis of interactions, composition and reuse of stateful aspects.

We model AOP through weaving by means of a dynamic monitor, which observes the execution of the program and inserts instructions according to execution states.

The relevant part of an execution for weaving is called the observable execution trace. It can be formally defined on the basis of a small-step semantics of the base programming language. The observable trace is a sequence of join points which are abstractions of the execution state of the program.

The primitive constituents of our aspect language are *basic rules* $C \rhd I$ where C is a *crosscut* and I an *insert*. Crosscuts are patterns matching join points whereas inserts are templates. The intuition behind a basic rule is that when the crosscut matches the current join point, it yields a substitution which is applied to the insert before executing it. The basic rule $\texttt{error}(m) \rhd \texttt{abort}()$ aborts the current execution when a join point is encountered which matches the call to the one parameter routine `error`.

Aspects combine basic rules using three operators: *prefixing* of basic rules to aspects, *choice* between two aspects and *repetition* of an aspect. Aspects match sequences of join points and they evolve according to the join points they match. For example, an aspect intended to log warning messages in a log file may wait for the log file to be opened and then store warnings repeatedly in that file. Such aspects are called *stateful*: a state is needed to represent their evolution.

Aspects are defined using the following grammar:

$$
\begin{array}{lcll}
A & ::= & \mu a.A & \text{; recursive definition} \quad (a \in \mathcal{R}ec) \\
& | & C \rhd I;\ A & \text{; prefixing} \\
& | & C \rhd I;\ a & \text{; end of sequence} \qquad (a \in \mathcal{R}ec) \\
& | & A_1 \ \square\ A_2 & \text{; choice}
\end{array}
$$

An aspect is either:

- A recursive definition.
- A sequence formed using the prefix operation $C \rhd I; X$, where X is an aspect or a variable. When the crosscut C matches the program point and its variables have a unique solution, we write $C\ j = \phi$ where ϕ is a substitution mapping the variables in C to their solution. The variables in C used in I are replaced by their solution and X becomes the aspect to be woven. Otherwise, we say that the crosscut does not match the program point and we write $C\ j = \texttt{fail}$.
- A choice construction $A_1 \ \square\ A_2$ which chooses the first aspect that matches a join point (the other is thrown away). If both match the same join point, A_1 is chosen.

For instance, a logging aspect which, after opening a log file, logs warnings but aborts program execution when an error occurs can be expressed as follows:

$$
\begin{array}{l}
\mathit{ErrLog} = \\
\quad \texttt{openLog}() \rhd \texttt{skip};\ \mu a.\texttt{warning}(m) \rhd \texttt{writeLog}(m);\ a \\
\qquad\qquad\qquad\qquad \square\ \texttt{error}(m) \rhd \texttt{abort}();\ a
\end{array}
$$

where the insert `skip` is the instruction doing nothing.

We consider only closed aspects, *i.e.,* with no free variables in inserts nor free $\mathcal{R}ec$ variables. To ensure that aspect are finite state, recursion occurs only as tail recursive calls. Aspects keep trying to match the join points of the execution trace and never terminate (note that the aspect *ErrLog* formally does not stop but that it causes the base program execution to be aborted).

Aspects addressing different issues, *e.g.,* different error handling strategies, security and profiling, are composed using a *parallel operator*. *Weaving* defines how matching of crosscuts and execution of inserts is interleaved with the base program's execution. Intuitively, the weaver takes a parallel composition of n aspects $A_1 \| \ldots \| A_n$ and performs the following steps at each join point:

- The applicable basic rules (whose crosscuts match the current join point) are determined by a function **sel**.

  For instance, in case of the aspect *ErrLog* defined above, **sel** yields the rule $\texttt{openLog}() \rhd \texttt{skip}$ at the `first` join point opening the log file. The empty set is yielded for, among others, all log-opening join points after the first one.

- All selected basic rules are applied (*i.e.,* their inserts executed) in no specific order.

  In the case of the first rule of *ErrLog*, nothing would be done because the insert `skip` is applied.

- The evolution of $A_1 \| \ldots \| A_n$ is computed by a function **next**.

  For *ErrLog* this means that after the first join point which opens the log file has been handled, the aspect to be considered (*i.e.,* the result of **next**) is

  $$
  \begin{array}{l}
  \mu a.\texttt{warning}(m) \rhd \texttt{writeLog}(m);\ a \\
  \quad \square\ \texttt{error}(m) \rhd \texttt{abort}();\ a
  \end{array}
  $$

  Note that the aspect *ErrLog* does not evolve from its initial state before encountering the join point that opens the log file.

- A standard execution step of the base program is performed, yielding a new current join point.

These steps are iterated with the new aspect and the next join point until the base program terminates.

More formally, the weaver makes use of **sel** which takes a composition of aspects and extracts the rules to apply at the current join point j.

$$
\begin{array}{lcll}
\textbf{sel}\ j\ (A_1 \| \ldots \| A_n) & = & (\textbf{sel}\ j\ A_1) \cup \ldots \cup (\textbf{sel}\ j\ A_n) & \\
\textbf{sel}\ j\ (\mu a.A) & = & \textbf{sel}\ j\ A & \\
\textbf{sel}\ j\ (C \rhd I;\ A) & = & \emptyset & \text{if } C\ j = \texttt{fail} \\
 & = & \{C \rhd I\} & \text{otherwise} \\
\textbf{sel}\ j\ (A_1 \ \square\ A_2) & = & \textbf{sel}\ j\ A_1 & \text{if } \textbf{sel}\ j\ A_1 \neq \emptyset \\
 & = & \textbf{sel}\ j\ A_2 & \text{otherwise}
\end{array}
$$

The evolution of an aspect after the application of a basic rule is described by the **next** function. It takes a composite aspect, the current join point and yields the aspect to be applied to the next join point. It makes use of the function **sel** which takes an aspect and extracts the rule to apply at the current join point j.

$$
\begin{array}{lcll}
\textbf{next}\ j\ (A_1 \| \ldots \| A_n) & = & (\textbf{next}\ j\ A_1) \| \ldots \| (\textbf{next}\ j\ A_n) & \\
\textbf{next}\ j\ (\mu a.A) & = & \textbf{next}\ j\ A[\mu a.A/a] & \\
\textbf{next}\ j\ (C \rhd I;\ A) & = & C \rhd I;\ A & \text{if } C\ j = \texttt{fail} \\
 & = & A & \text{if } C\ j = \phi \\
\textbf{next}\ j\ (A_1 \ \square\ A_2) & = & \textbf{next}\ j\ A_1 & \text{if } \textbf{sel}\ j\ A_1 \neq \emptyset \\
 & = & \textbf{next}\ j\ A_2 & \text{if } \textbf{sel}\ j\ A_2 \neq \emptyset \\
 & = & (A_1 \ \square\ A_2) & \text{otherwise}
\end{array}
$$

The woven execution performed relative to a composite aspect A is formalized in Figure 1. The entry and exit of a program are denoted by two special join points: $\downarrow$ and $\uparrow$,

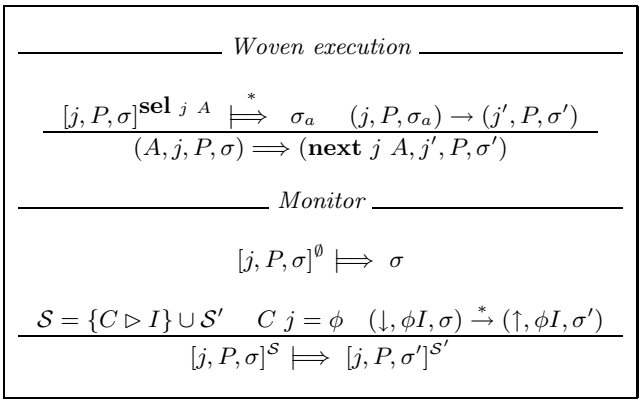

Figure 1: Weaving

respectively. The transition relation $\to$ represents the standard execution. Let σ_0 be the initial state, the observable execution trace of a program P is of the form:

$$(\downarrow, P, \sigma_0) \to \ldots \to (j_i, P, \sigma_i) \to \ldots$$

If the reduction terminates, there exists a σ_n such that $(\downarrow, P, \sigma_0) \overset{*}{\to} (\uparrow, P, \sigma_n)$, where $\overset{*}{\to}$ denotes the transitive, reflexive closure of $\to$. The woven execution $\Longrightarrow$ is defined by the application of the monitor followed by a standard execution step. It yields the aspect (**next** j A) to be applied to the following join point. At each join point, the applicable rules are selected (**sel** j A). The monitor (relation $\Longmapsto$) applies the selected rules in no specific order: if the crosscut of the current rule matches the current join point, the corresponding substitution is applied to the insert and ϕI is executed. To end the discussion of the weaver, note that stateful aspects are implementable efficiently using static analysis and transformation techniques (see, [2]).

Two distinct aspects are said to *interact* when they match the same join point. Two aspects are *independent* if their crosscuts never match the same join point simultaneously. Independence of two aspects ensures that their parallel composition is well-defined: they can be woven in any order. To the contrary, dependent (*i.e.,* interacting) aspects require the programmer to resolve the interactions by changing aspects or making the composition more precise.

Consider, for example, the aspect $A_{encryption}$ that matches some method calls and encodes their argument and an aspect $A_{logging}$ that logs some method calls. If some method calls are matched by both $A_{encryption}$ and $A_{logging}$, the aspects interact and their parallel composition is not well-defined. In this case, since $A_{encryption} \| A_{logging}$ does not specify any execution order for inserts, weaving is non-deterministic.

One main objective of the present work is to generalize the techniques for interaction analysis and conflict resolution introduced in [4]. Furthermore, we are interested in extending the framework by means for the more expressive definition of aspects. Concretely, we present three contributions in the following: introduction of inter-crosscut variables generalizing the aspect language suitable for static analysis, generalized means for aspect composition and conflict resolution, and new means for the expression of applicability requirements of aspects which support reuse of aspects.

In many cases, information must be passed between crosscuts of a complex aspect. Suppose, for example, that $A_{logging}$ should log only file deletions referring to the user currently logged in. (i.e., between calls to `login`(*uid*) and `logout`()). The aspect must wait for a login, record the corresponding user identity (*uid*), and log calls referring to this *uid* (e.g. `rm`($uid, file$)). When the session ends (`logout`() occurs), the aspect proceeds by waiting for the next login and so on. The identity of the current user (*uid*) has to be passed between crosscuts and this is done using pattern variables. Section 3 presents an expressive aspect language featuring pattern variables and the associated interaction analysis.

In Section 4, we introduce a *sequence composition operator* for aspects. This sequence operator explicitly uses a "termination crosscut": if this crosscut matches, the first aspect in the sequence is stopped and the second is activated. The operator gives rise to a flexible notion of scope of aspects which enables the scope of an aspect to be delimited by execution events. We also introduce a set of composition operators for the adaptation, *i.e.,* transformation, of aspects. These *composition adaptors* specify transformations on inserts of aspects. They are therefore highly useful to resolve conflicts in non-deterministic aspects resulting from a parallel composition. For example, it is easy to define the composition operator $\|_{\overline{seq}}$ (resp. $\|_{\overline{fst}}$) which sequentializes inserts (resp. applies only the first insert) at each interaction. By composing the logging and encryption aspects introduced before we can then resolve conflicts using these operators:

- $A_{logging} \|_{\overline{seq}} A_{encryption}$ generates logs for super users by logging method calls with original arguments,
- $A_{encryption} \|_{\overline{fst}} A_{logging}$ generates logs for basic users where the encrypted methods do not appear.

In general, an aspect is not valid for all base programs, but only for some because it relies on certain implicit context conditions. For example, in our previous description of $A_{logging}$, we implicitly assumed that sessions were non-nested (otherwise, `logout`() would not necessarily mean leaving the top-level session). We show in Section 5 that such conditions can be made explicit in our framework as *requirements* on base programs. These requirements define which base programs can correctly be woven with an aspect. Such requirements can be expressed as a companion aspect, i.e. as sequences of join points. For example, non-nested sessions can be specified by an aspect checking that no `login` call occurs between each `login`(*uid*) and `logout`(). We also show how aspect requirements can be used to make interaction analysis more precise. Finally, we give evidence that requirements support *reuse of aspects* by checking compatibility of aspects *w.r.t.* requirements defining use contexts (similar to how pre- and post-conditions in programming by contract are used to support software reuse).

In this section, we present an expressive aspect language and the associated interaction analysis. Technically, we extend the base framework by inter-crosscut pattern variables,

which permits to define aspects more precisely and to avoid detection of spurious conflicts.

As in the base framework, aspects are regular expressions (defined using the grammar A introduced in the previous section) of rules of the form: $C \rhd I$ where C denotes a crosscut and I an insert.

Crosscuts are built from terms, that is to say, finite trees of the form:

$$T ::= \mathtt{f}\ T_1 \dots T_n \mid x$$

where $\mathtt{f}$ is an n-ary ($n \geq 0$) symbol and $x \in \mathcal{V}ars$ is a pattern variable.

Crosscuts are made of conjunctions, disjunctions and negations of equations on terms:

$$C ::= v \doteq T \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C \qquad v \in \mathcal{V}ars$$

Assuming a special variable $\bullet$ denoting the current join point, the application of a crosscut to a join point $C\ j$ amounts to solving the formula obtained by substituting j for $\bullet$ in C ($C\ j = C[j/\bullet]$). There exists an algorithm to find equations solving such formulas [3]. It is used by the interaction analysis of Section 3.2.

We write $def(C)$ for the set of variables ($\subset \mathcal{V}ars$) occurring as *lhs* of equations in C. These variables are defined by the equations of C and can be used in the insert or later on in the aspect. We write *false* for the crosscut which does not match any join point and *true* for the crosscut that matches all join points. Let z be a fresh variable then *true* can be defined by the crosscut $z \doteq z$ and *false* by $\neg(z \doteq z)$.

An insert I is a term as defined above. The intuition behind a rule $C \rhd I$ is that when the crosscut matches the current join point, i.e., $C\ j = \phi$, then ϕI is executed. Hence, the insert can only use variables defined in C *or a previous crosscut*. The special insert `skip` represents an instruction doing nothing.

An equation $z \doteq T$ defines the variable z whose scope ends at the subsequent (re)definition of z. Variables can be used to pass information between crosscuts. For example, an aspect counting the number of calls of the first method ever called by the base program can be expressed using inter-crosscut variables as follows:

$$\begin{array}{rcl} A_1 & = & \bullet \doteq \mathtt{call}\ x\ \wedge\ f \doteq x\ \rhd\ \mathtt{first.set(1)}; \\ & & (\mu a.\bullet \doteq \mathtt{call}\ f\ \rhd\ \mathtt{first.inc}();\ a) \end{array}$$

To simplify the notation, we write $\hat{}T$ for $\bullet \doteq T$ and $\overline{x}$ to define and use a variable in a pattern. This syntactic sugar can be suppressed from a crosscut C using the following rules:

$$\begin{array}{rcl} C & = & C[(\bullet \doteq T)/\hat{}T] \\ C & = & C[z/\overline{x}]\ \wedge\ x \doteq z \quad \text{with } z \text{ a fresh variable} \end{array}$$

With these conventions, the aspect A_1 can be written:

$$\hat{}\mathtt{call}\ \overline{f}\ \rhd\ \mathtt{first.set(1)};\ (\mu a.\hat{}\mathtt{call}\ f\ \rhd\ \mathtt{first.inc}();\ a)$$

Note that without inter-crosscut variables, such an aspect would require book-keeping code in inserts (*e.g.,* memorizing the name of the first method for comparison in further calls) which would be executed for all calls.

The semantics of aspects is given by the weaver as described in Section 2.1. Here, we focus on the differences brought forth by inter-crosscut variables.

When a crosscut matches the current join point, the substitution found is applied to the rest of the aspect. This means that variable bindings from the preceding basic aspect must be passed to the next one. In order to account for this, the function **next** presented in the previous section must be modified as follows:

$$\begin{array}{rcll} \mathbf{next}\ j\ (C \rhd I;\ A) & = & C \rhd I;\ A & \text{if } C\ j = \mathtt{fail} \\ & = & \phi A & \text{if } C\ j = \phi \end{array}$$

Here, the application of a substitution ϕ to an aspect A is defined as follows:

$$\begin{array}{lcll} \phi(\mu a.A) & = & \mu a.\phi A & \\ \phi(A_1 \,\square\, A_2) & = & (\phi A_1 \,\square\, \phi A_2) & \\ \phi(C \rhd I;\ A) & = & \phi' C \rhd \phi' I;\ \phi' A & \text{with } \phi' = \phi \backslash def(C) \end{array}$$

Applying a substitution to a crosscut C thus amounts to applying it to each term occurring in C. Furthermore, variable redefinitions performed in prefix operations are observed by using the substitution $\phi \backslash def(C)$, which denotes the restriction of ϕ to the variables not (re)defined in C, in the last case.

Our goal is to keep the writing of aspects as independent as possible from their composition. Furthermore, we do not want to compel the programmer to specify a (useless) order of application for independent aspects. In our approach, aspects are first written and composed in parallel. Then, interactions are detected using static analysis and resolved by making the composition more specific. This way, the composition of aspects is specified separately and only when needed.

There are several sufficient properties ensuring the absence of interactions. We focus here on *strong independence* that does not depend on the program to be woven: strongly independent aspects do not interact regardless of the base program or the inserts which can be woven. Strong independence thus does not have to be checked after each program modification. (A more precise variant of the interaction analysis which takes into account the base program and new behavior introduced by other inserts has been presented in [4]. We expect that the extension by variables carries over smoothly to that case as well).

The algorithm to check strong independence of aspects is based on the laws (which can be proved correct *w.r.t.* the weaving semantics) shown in Figure 2. The algorithm, which is similar to the algorithm for finite-state product automata, propagates and suppresses parallel operators. It terminates due to the finite-state nature of our aspects (the *(un)fold* law is used to fold already encountered aspects). If the composition of aspects can be rewritten into a sequential and deterministic aspect then the aspects are independent and weaving is well defined. Otherwise, nondeterministic inserts occur in the resulting aspect. They represent conflicts to be resolved by the techniques presented in Section 4.

To avoid name capture problems, we assume that the aspects of a parallel composition use disjoint sets of variables.

[*(un)fold*]	$\mu a.A = A[\mu a.A/a]$	
[*assoc*]	$(A_1 \,\square\, A_2) \,\square\, A_3 = A_1 \,\square\, (A_2 \,\square\, A_3)$	
[*commut*]	$(C_1 \rhd I_1;\ A_1) \,\square\, (C_2 \rhd I_2;\ A_2) = (C_2 \rhd I_2;\ A_2) \,\square\, (C_1 \rhd I_1;\ A_1)$	; if $C_1 \wedge C_2$ has no solution
[*elim*$_1$]	$C \rhd I = \mathit{false} \rhd I$	; if C has no solution
[*elim*$_2$]	$(\mathit{false} \rhd I;\ X) \,\square\, A = A$	
[*elim*$_3$]	$\mathit{false} \rhd I;\ X = \mathit{false} \rhd I;\ a$	; $a \in \mathcal{R}ec$
[*skip*]	$(\texttt{skip} \bowtie I) = (I \bowtie \texttt{skip}) = I$	
[*priority*]	$(C_1 \rhd I_1;\ A_1) \,\square\, (C_2 \rhd I_2;\ A_2) = (C_1 \rhd I_1;\ A_1) \,\square\, (C_2 \wedge \neg\psi C_1 \rhd I_2;\ A_2)$	; ψ is a renaming
[*seq*]	$C \rhd I;\ (\square_{i=1..n} C_i \rhd I_i;\ A_i) = C \rhd I;\ (\square_{i=1..n} C_i[C] \rhd I_i;\ A_i)$	; from $\mathcal{V}ars$ to fresh variables
	with $C_i[C] = C_i \wedge \psi C \ \bigwedge_{x \in def(C)} \psi x \doteq x$	
[*propag*]	let $A = (C_1 \rhd I_1;\ A_1) \,\square\, \ldots \,\square\, (C_n \rhd I_n;\ A_n)$ and $A' = (C'_1 \rhd I'_1;\ A'_1) \,\square\, \ldots \,\square\, (C'_m \rhd I'_m;\ A'_m)$ then $A \parallel A' = \square^{j=1..m}_{i=1..n} C_i \wedge C'_i \rhd (I_i \bowtie I'_j);\ (A_i \parallel A'_j)$ $\square_{i=1..n} C_i \rhd I_i;\ (A_i \parallel A')$ $\square_{j=1..m} C'_j \rhd I'_j;\ (A \parallel A'_j)$	

Figure 2: Laws for aspects

We illustrate the analysis and explain the rules on an example. Let A_1 be the aspect introduced in Section 3.1 and A_2 the aspect

$$\hat{}\texttt{call}\ \overline{g} \rhd \texttt{skip};\ (\mu a.\hat{}\texttt{call}\ h \wedge \neg(h \doteq g) \rhd \texttt{others.inc}();\ a)$$

counting the number of calls to all methods except the first one called. Intuitively, these aspects do not interact and their naive parallel composition is well defined. This is established by the analysis using the rules of Figure 2 as follows.

Let
$C = \hat{}\texttt{call}\ \overline{f} \wedge \hat{}\texttt{call}\ \overline{g}$
$C' = \hat{}\texttt{call}\ f \wedge \hat{}\texttt{call}\ h \wedge \neg(h \doteq g)$
$A'_1 = \mu a.\hat{}\texttt{call}\ f \rhd \texttt{first.inc}();\ a$
$A'_2 = \mu a.\hat{}\texttt{call}\ h \wedge \neg(h \doteq g) \rhd \texttt{others.inc}();\ a$

then

$A_1 \parallel A_2$
$= \hat{}\texttt{call}\ \overline{f} \wedge \hat{}\texttt{call}\ \overline{g} \rhd (\texttt{skip} \bowtie \texttt{first.set(1)});\ (A'_1 \parallel A'_2)$
$\square\ \hat{}\texttt{call}\ \overline{f} \rhd \texttt{first.set(1)};\ (A'_1 \parallel A'_2)$
$\square\ \hat{}\texttt{call}\ \overline{g} \rhd \texttt{skip};\ (A'_1 \parallel A'_2)$ [*propag*]

The parallel composition (product) of aspects is performed by [*propag*] which propagates the parallel operator inside the aspect definition. It produces a sequence of choices made of all the possible pairs of crosscuts from A and A' and all the single crosscuts of A and A' independently. Conflicts are represented using the non-deterministic function $(I_1 \bowtie I_2)$ which returns either $I_1;I_2$ or $I_2;I_1$ (where ";" denotes the sequencing operator of the programming language). In the example, $(\texttt{skip} \bowtie \texttt{first.set(1)})$ is not a true conflict since the inserts commute.

$= C \rhd \texttt{first.set(1)};\ (A'_1 \parallel A'_2)$
$\square\ \hat{}\texttt{call}\ \overline{f} \wedge \neg C \rhd \texttt{first.set(1)};\ (A'_1 \parallel A'_2)$
$\square\ \hat{}\texttt{call}\ \overline{g} \wedge \neg(\hat{}\texttt{call}\ \overline{f} \wedge \neg C) \rhd \texttt{skip};\ (A'_1 \parallel A'_2)$
[*skip*], [*priority*], [*assoc*]

The [*skip*] rule is an instance of the simplification that can be done by taking into account the semantics of inserts. whenever two inserts commute. Many similar laws could be conceived. In particular, whenever two inserts I_1 and I_2 commute, $(I_1 \bowtie I_2)$ is equivalent to $I_1;I_2$ (or $I_2;I_1$). The [*priority*] rule accounts for the priority rules implicit in the choice operator. This makes the analysis more precise by allowing simplifications. The renaming is needed to avoid clashes between variables names occurring in the two crosscuts (the renaming needs not be applied to I_2 because variables are not visible in a sibling argument of a choice). The law [*assoc*] (as well as [*commut*] and [*(un)fold*]) serves to put aspects on a form appropriate for further rewriting.

$= C \rhd \texttt{first.set(1)};\ (A'_1 \parallel A'_2)$
$\square\ \mathit{false} \rhd \texttt{first.set(1)};\ (A'_1 \parallel A'_2)$
$\square\ \mathit{false} \rhd \texttt{skip};\ (A'_1 \parallel A'_2)$ [*elim*$_1$]

The law [*elim*$_1$] uses the algorithm of [3] to check if a crosscut has no solution in which case the crosscut is replaced by *false*. In the example, if the join point does not match C it is not a call; it cannot match $\texttt{call}\ \overline{f}$ or $\texttt{call}\ \overline{g}$ either.

$= C \rhd \texttt{first.set(1)};\ (A'_1 \parallel A'_2)$ [*elim*$_2$]

The law [*elim*$_2$] (as [*elim*$_3$]) removes unreachable parts of an aspect.

$= C \rhd \texttt{first.set(1)};$
$(\mu a.C' \rhd (\texttt{first.inc}() \bowtie \texttt{others.inc}());\ a$
$\square\ \hat{}\texttt{call}\ f \rhd \texttt{first.inc}();\ a$
$\square\ \hat{}\texttt{call}\ h \wedge \neg(h \doteq g) \rhd \texttt{others.inc}();\ a)$ [*propag*]

The expression has been unfolded ([*(un)fold*]), the parallel operator has been propagated using [*propag*] and suppressed by folding the expression back ([*(un)fold*]).

$= C \rhd \texttt{first.set(1)};$
$(\mu a.C'[C] \rhd (\texttt{first.inc}() \bowtie \texttt{others.inc}());\ a$
$\square\ (\hat{}\texttt{call}\ f)[C] \rhd \texttt{first.inc}();\ a$
$\square\ (\hat{}\texttt{call}\ h \wedge \neg(h \doteq g))[C] \rhd \texttt{others.inc}();\ a)$ [*seq*]

The [*seq*] rule serves to propagate the constraints on variables introduced by a crosscut. The propagation renames all the variables of the crosscut (including $\bullet$) by fresh variables to avoid name clashes. For our example, C

$$\bullet \doteq \texttt{call}\ x \ \wedge\ f \doteq x \ \wedge\ \bullet \doteq \texttt{call}\ y \ \wedge\ g \doteq y$$

has its variables renamed into

$$z \doteq \texttt{call}\ z_1 \ \wedge\ z_2 \doteq z_1 \ \wedge\ z \doteq \texttt{call}\ z_3 \ \wedge\ z_4 \doteq z_3$$

The useful information is passed by binding the variables in $def(C)$ to their renaming. In our example,

$$z_2 \doteq f \ \wedge\ z_4 \doteq g$$

Therefore,

$$C'[C] \;=\; \hat{}\texttt{call}\ f \ \wedge\ \hat{}\texttt{call}\ h \ \wedge\ \neg(h \doteq g) \ \wedge\ z \doteq \texttt{call}\ z_1 \ \wedge\ z \doteq \texttt{call}\ z_3 \ \wedge\ z_2 \doteq z_1 \ \wedge\ z_4 \doteq z_3 \ \wedge\ z_2 \doteq f \ \wedge\ z_4 \doteq g$$

which is *false*: C' implies $f \neq g$ whereas the information brought by C implies $f = g$. The associated rule can be removed using $[elim_1]$ and $[elim_2]$.

$$= \; C \rhd \texttt{first.set(1)};\ (\mu a.(\hat{}\texttt{call}\ f)[C] \rhd \texttt{first.inc}();\ a \ \square\ (\hat{}\texttt{call}\ h \wedge \neg(h \doteq g))[C] \rhd \texttt{others.inc}();\ a)$$

So, $A_1 \parallel A_2$ has been rewritten into a deterministic, sequential aspect. The aspects A_1 and A_2 are strongly independent. Without inter-crosscut variables, A_1 would have been expressed by book-keeping code such as:

$$\hat{}\texttt{call}\ x \rhd \texttt{first.set(1); name = x;}$$
$$\mu a.\hat{}\texttt{call}\ y \rhd \texttt{if y == name then first.inc() else skip};\ a$$

This aspect matches *all* calls. It would be found to interact with any other aspect matching calls (like A_2).

The parallel operator $\parallel$ enables different aspects to be combined along with the analysis of their interactions. In this section, we generalize aspect composition in two different ways while preserving feasibility of static analysis of interactions. Using these extensions, we then define expressive support for conflict resolution of interacting aspects.

In Section 3, we have considered three different operators allowing to combine aspects: recursion (μ), choice ($\square$) and parallel composition ($\parallel$). However, prefixing ($C \rhd I;\ X$, where X is an aspect or a recursion variable) is only defined starting with a basic rule.

Our first extension consists in the new composition $A_1 -C\rightarrow A_2$ which behaves as the aspect A_1 until an event matches the crosscut C; in this case, A_1 is stopped and the aspect A_2 is started. This sequence operator can be formally defined as

$$A_1 -C\rightarrow A_2 = T[\![A_1]\!]^C_{A_2}$$

where T is the following transformation which eliminates sequencing between aspects, thus yielding a "standard" aspect.

$$\begin{array}{lcl} T[\![\mu a.A]\!]^{C'}_{A'} & = & \mu a.T[\![A]\!]^{C'}_{A'} \\ T[\![C \rhd I;\ A]\!]^{C'}_{A'} & = & C \wedge C' \rhd I;\ A' \\ & & \square\ C' \rhd \texttt{skip};\ A' \\ & & \square\ C \rhd I;\ T[\![A]\!]^{C'}_{A'} \\ T[\![a]\!]^{C'}_{A'} & = & a \\ T[\![A_1 \ \square\ A_2]\!]^{C'}_{A'} & = & T[\![A_1]\!]^{C'}_{A'} \ \square\ T[\![A_2]\!]^{C'}_{A'} \end{array}$$

Here, the interesting case is the transformation of prefixing $C \rhd I;\ A$, which starts with a basic rule. Three cases must be distinguished. First, when the current join point matches the "terminating" crosscut C' as well as the crosscut C of the basic rule, the insert I is executed, the current aspect is terminated and A' is started.* Second, when only C' is matched, no insert (*i.e.,* `skip`) is executed and A' becomes the new aspect. Finally, if C' does not match but C does, I is inserted and the transformed version of the aspect A is executed.

As a simple example, consider the following two definitions:

$$\begin{array}{lcl} E_1 & = & \mu a.\hat{}\texttt{error}(x) \rhd \texttt{beep}();\ a \\ E_2 & = & \mu a.\hat{}\texttt{error}(\overline{m}) \rhd \texttt{writeLog}(m);\ a \end{array}$$

E_1 and E_2 define two aspects for error handling: the former marks errors by a beep while the latter logs error messages. The second error handling strategy obviously is only reasonable if the log file has previously been created. In order to account for that need we could define error handling as $E_1 -\texttt{createLog}()\rightarrow E_2$, which ensures that errors produce beeps up to the point when a log file is created; from that point on error messages are logged.

The composition operator $A_1 -C\rightarrow A_2$ can be naturally interpreted as "A_1 until C then A_2", i.e., an operator defining a flexible notion of *scope*, which delimits the scope of A_1 based on event occurrences. We could define many different other operators supporting such flexible scoping, e.g., "A until C" as $A -C\rightarrow$ *never* (where *never* denotes an aspect which matches no event), "A from C" as $C \rhd \texttt{skip};\ A$, and also more complex ones such as "A should be enabled every other crosscut". Note that the expressions constructed from such operators can be freely composed with one another.

The second extension geared towards aspect composition we propose enables adaptation, *i.e.,* transformation, of parallel compositions. Technically, we purport *composition adaptors*, operators O constructed using the following grammar:

$$\begin{array}{lcll} O & ::= & \mu a.O & \textit{; recursive definition} \\ & | & C \rhd F;\ O & \textit{; prefixing} \\ & | & C \rhd F;\ a & \textit{; end of sequence} \\ & | & O_1 \ \square\ O_2 & \textit{; choice} \\ \\ F & ::= & (U \oplus B) & \textit{; pair of transformers} \\ U & ::= & id \mid skip & \textit{; unary transformers} \\ B & ::= & \bowtie \mid seq \mid fst \mid snd \mid skip & \textit{; binary transformers} \end{array}$$

In a pair of transformers $(u \oplus b)$, $u : I \rightarrow I$ and $b : I \times I \rightarrow I$ are unary and binary transformers of inserts, respectively. The function *skip* is the constant function yielding `skip`, $seq(I_1, I_2)$ yields $I_1; I_2$, and *fst* (*snd*) the first (second) argument. Many other unary and binary functions could be considered.

We note composition operators which use a composition adaptor O as $\parallel_O$. Intuitively, an adapted aspect composition $A = A_1 \parallel_O A_2$ evolves throughout the composition the same way as the plain aspect composition $A_1 \parallel A_2$. The different composition functions $(u \oplus b)$ occurring in a basic rule r of the composition adaptor O are applied to inserts of A_1 and A_2 at join points matching the crosscut of r and at least one of the corresponding crosscuts of A_1 and A_2. If only one of the two aspects matches, u is applied to the corresponding insert, whereas b is applied when the two aspects interact. This way, adaptors allow the selective modification of inserts generated by a composition. Note that the plain parallel composition can be expressed as the following composition operator:

$$\parallel = \parallel_X \quad with \quad X = \mu a.true \rhd (id \oplus \bowtie);\ a$$

*This case means that our operator includes the behavior of the left aspect at "termination points"; applying `skip` at those points (*i.e.,* excluding the behavior) would give a different sequence operator.

The adaptor X matches all join points and composes conflicting inserts using $\bowtie$.

An adapted composition ($A_1 \|_O A_2$ say) can be analyzed for strong independence by taking into account the evolution of the adaptation operator O along with those of the aspects A_1 and A_2. At join points where O and A_1 or A_2 interact, the adaptation function defined in the corresponding insert of O is applied to the corresponding insert of the parallel composition of the two aspects.

The interaction analysis remains the same except that the rule [*propag*] of Figure 2 is replaced by the rule shown in Figure 3. As the original rule, the new one considers all conjunctive terms constructed from combinations of all crosscuts of the aspects involved. In contrast to the original rule, propagation of adapted compositions first subjects all pairs and individual terms built from A_1 and A_2 to the crosscuts of O: if one of these crosscuts C_k^O matches, the transformers (u_k, b_k) are applied to the corresponding inserts A_i and A_j. The prioritization of the choice operator ensures that the transformers are applied whenever possible.

Composition operators and adaptors can be used as a means to resolve conflicts of interacting aspects. More precisely, a programmer can use them to get rid of the nondeterminism introduced by aspect compositions.

The analysis of strong independence (Section 2.1) returns a sequential aspect. The occurrences of rules of the form $C \rhd I_1 \bowtie I_2$ indicate potential interactions. These interactions can be resolved one by one. For each $C \rhd I_1 \bowtie I_2$, the programmer may replace each rule $C \rhd I_1 \bowtie I_2$ by $C \rhd I_3$ where I_3 is a new insert which combines I_1 and I_2 in some way. This option is flexible but can be tedious. Instead of writing a new insert for each conflict, the programmer may use composition adaptors to indicate how to compose inserts at the aspect level.

Composition operators can be used to this end. For example, the aspects E_1 and E_2 introduced above for error handling interact at all erroneous states if composed in parallel. The sequence $E_1 -\texttt{createLog}()\rightarrow E_2$ can then be used to resolve interactions by ensuring that only one aspect is active at a time.

Composition adaptors can be applied as an expressive means to resolve conflicts. The binary transformers b used in an adapted composition allow conflicts to be resolved by means of, for instance, orderings between inserts (*e.g.,* using the binary function *seq*) or ignoring some insert (*e.g.,* using *fst*).

As an example, let us consider the following definitions:

$$\begin{array}{lcl} E_2 & = & \mu a.\hat{}\texttt{error}(\overline{m}) \rhd \texttt{writeLog}(m);\ a \\ E_3 & = & \hat{}\texttt{logout}() \rhd \texttt{closeLog}(); \\ & & \quad \mu a.\hat{}\texttt{error}(x) \rhd \texttt{beep}();\ a \end{array}$$

E_2 is the same aspect as above (expressing that errors should be written to a log file). E_3 closes the log file as part of a logout and afterward errors are only marked by beeps The simple composition $E_2 \parallel E_3$ yields interactions at all error join points occurring after logouts.

We can resolve these interactions using the adapted composition $E_2 \|_O E_3$, where

$$\begin{array}{lcl} O & = & \hat{}\texttt{logout}() \rhd (id, skip); \\ & & \quad \mu a.\hat{}\texttt{error}(y) \rhd (id, snd);\ a \end{array}$$

This composition yields the aspect:

$$\begin{array}{l} \mu a.\hat{}\texttt{error}(\overline{m}) \rhd \texttt{writeLog}(m);\ a \\ \quad \Box\ \hat{}\texttt{logout}() \rhd \texttt{closeLog}(); \\ \qquad \mu b.\hat{}\texttt{error}(x) \rhd \texttt{beep}();\ b \end{array}$$

where the propagation rule for adapted composition causes conflicts appearing in the nested loop (using recursion variable b) to be resolved using *snd*. After `logout()`, the stateful operator O eliminates inserts from E_2 at each conflict (*i.e.,* each occurrence of `error`).

A useful class of composition operators are stateless operators, denoted $\|_{\overline{f}}$, which use a composition adaptor defined as

$$\mu a.true \rhd (id \oplus f);\ a$$

These operators use the same function for resolving all conflicts. As an example, let us consider two error handling aspects: A_{abort} matches every `error`(m) and may abort (some) programs depending on the message m; $A_{correct}$ matches every `error`(m) and may correct (some) errors depending on m. Parallel compositions of these two aspects interact on *all* errors, but their inserts abort on *some* errors and correct *some* errors. Different strategies can be used to resolve conflicts:

- $A_{abort} \|_{\overline{seq}} A_{correct}$ only allows the correction of non-fatal errors, *e.g.,* to avoid corrections potentially leading to later problems.
- $A_{correct} \|_{\overline{seq}} A_{abort}$ gives priority to error correction.

In general, an aspect definition is reasonable for only some base programs. In this section, we propose an extension to the basic framework in order to make this fact explicit by defining validity domains in terms of the sequences of join points that base programs are required to generate. These explicit requirements allow us to define a notion of independence which is weaker than strong independence: taking into account the expected behavior of base programs eliminates numerous spurious conflicts. Moreover, such contextual information makes aspect oriented programming safer: requirements can be checked when an aspect is to be woven with a specific base program, *i.e.,* before execution. Finally, we show that explicit validity domains make aspect definitions more reusable: provided that an arbitrary base program satisfies the necessary requirements, it is guaranteed that the corresponding aspect can be woven with it.

As a simple example why requirements on the intended validity domain of aspects are useful, let us consider the following aspect *Log*:

$$\begin{array}{l} Log\ = \mu a_1.\hat{}\texttt{call}(\texttt{login}(\overline{uid}))\ \rhd\ \texttt{addLog}(uid); \\ \qquad \mu a_2.\ \hat{}\texttt{call}(\texttt{logout}())\ \rhd\ \texttt{skip};\ a_1 \\ \qquad\quad \Box\ \hat{}\texttt{call}(\texttt{read}(\overline{f}))\ \rhd\ \texttt{addLog}(f);\ a_2 \end{array}$$

This aspect is intended to log file accesses *during* sessions (*i.e.,* from a call to `login` to the next call to `logout`). Moreover, the user identity *uid* is logged at the beginning of a session. When the base program performs the following sequence of actions:

```
login("Bob"); read("file1"); login("Sam");
  read("file2"); logout(); logout();
```

$$
\begin{array}{lll}
\text{let} & A & = (C_1 \rhd I_1;\ A_1) \,\square\, \ldots \,\square\, (C_m \rhd I_m;\ A_m) \\
\text{and} & A' & = (C'_1 \rhd I'_1;\ A'_1) \,\square\, \ldots \,\square\, (C'_n \rhd I'_n;\ A'_n) \\
\text{and} & O & = (C^O_1 \rhd (u_1 \oplus b_1);\ O_1) \,\square\, \ldots \,\square\, (C^O_o \rhd (u_o \oplus b_o);\ O_o)
\end{array}
$$

$$
\begin{array}{lll}
\text{then} & A \parallel_O A' = & \square_{i=1..m,j=1..n,k=1..o}\, C_i \wedge C'_j \wedge C^O_k \rhd b_k(I_i, I'_j);\ (A_i \parallel_{O_k} A'_j) \\
& & \square\ \square^{j=1..m}_{i=1..n}\, C_i \wedge C'_j \rhd (I_i \bowtie I'_j);\ (A_i \parallel_O A'_j) \\
& & \square\ \square^{k=1..o}_{i=1..n}\, C_i \wedge C^O_k \rhd u_k(I_i);\ (A_i \parallel_{O_k} A') \\
& & \square\ \square_{i=1..n}\, C_i \rhd I_i;\ (A_i \parallel_O A') \\
& & \square\ \square^{k=1..o}_{j=1..m}\, C'_j \wedge C^O_k \rhd u_k(I'_j);\ (A \parallel_{O_k} A'_j) \\
& & \square\ \square_{j=1..m}\, C'_j \rhd I'_j;\ (A \parallel_O A'_j)
\end{array}
$$

Figure 3: Propagation of composition operators

the aspect logs that `file1` *and* `file2` are accessed by `Bob`. Indeed, the *Log* aspect ignores the second `login` when it is looking for `logout` or `read`. Imagine that this behavior is deemed not correct and *Log* should only be woven with base programs implementing *non*-nested sessions. We propose to specify such a requirement using an additional aspect:

$$
\begin{array}{l}
Flat = \mu a.\ \hat{}\texttt{call}(\texttt{login}(\overline{uid})) \rhd \texttt{skip};\\
\qquad (\ \hat{}\texttt{call}(\texttt{logout}()) \rhd \texttt{skip};\ a \\
\qquad\ \square\ \hat{}\texttt{call}(\texttt{login}(x)) \rhd \texttt{abort}();\ a) \\
\qquad \square\ \hat{}\texttt{call}(\texttt{logout}()) \rhd \texttt{abort}();\ a
\end{array}
$$

The aspect *Flat* monitors the desired requirement: it matches sequences of flat sessions, doing nothing in such cases. The rules $\hat{}\texttt{call}(\texttt{login}(x)) \rhd \texttt{abort}()$ and $\hat{}\texttt{call}(\texttt{logout}()) \rhd \texttt{abort}()$ stop execution when an unexpected join point occurs, that is when a `login` (resp. a `logout`) occurs within (resp. outside) a session.

When aspects interact it is possible that their interactions exclusively stem from execution traces which never occur when these aspects are applied to concrete base programs. We propose to take aspect requirements into account in order to get a more precise interaction analysis. This analysis lies between strong independence analysis (which shows that aspects never interact regardless of the base program to which they are applied) and the weak independence analysis introduced in [4] (which shows that aspects do not interact for a specific base program). By taking into account requirements, the contextual interaction analysis proofs that aspects do not interact for a set of related base programs.

Reconsidering the previous example, the aspect *Log* should be used only in the context of flat sessions. The parallel composition $Log \| Flat$ returns an instrumented version of *Log* that either terminates the execution when the requirement is violated or generates logs otherwise. Simplifying the parallel composition we get:

$$
\begin{array}{l}
FlatLog = Log \| Flat = \\
\quad \mu a_1.\hat{}\texttt{call}(\texttt{login}(\overline{uid})) \ \rhd\ \texttt{addLog}(uid); \\
\qquad (\mu a_2.\ \hat{}\texttt{call}(\texttt{logout}()) \ \rhd\ \texttt{skip};\ a_1 \\
\qquad\quad \square\ \hat{}\texttt{call}(\texttt{read}(\overline{f})) \ \rhd\ \texttt{addLog}(f);\ a_2 \\
\qquad\quad \square\ \hat{}\texttt{call}(\texttt{login}(x)) \rhd\ \texttt{abort}();\ a_2) \\
\qquad \square\hat{}\texttt{call}(\texttt{logout}()) \ \rhd\ \texttt{abort}();\ a_1
\end{array}
$$

Let us now consider the complementary aspect *SULog* that logs super user calls to `read` *between* sessions:

$$
\begin{array}{l}
SULog = \mu a_1.\hat{}\texttt{call}(\texttt{read}(\overline{f})) \ \rhd\ \texttt{addSULog}(f);\ a_1 \\
\qquad \square(\hat{}\texttt{call}(\texttt{login}(\overline{uid})) \ \rhd\ \texttt{skip}; \\
\qquad\quad \hat{}\texttt{call}(\texttt{logout}()) \ \rhd\ \texttt{skip};\ a_1)
\end{array}
$$

In general, *Log* and *SULog* are not strongly independent. However, *SULog* also requires flat sessions. So, interaction analysis can take into account these requirements by considering the parallel composition of the instrumented versions of both aspects:

$$
\begin{array}{l}
(SULog \| Flat) \| (Log \| Flat) = (SULog \| Log) \| Flat = \\
\quad \mu a_1.\hat{}\texttt{call}(\texttt{read}(\overline{f})) \ \rhd\ \texttt{addSULog}(f);\ a_1 \\
\qquad \square\ \hat{}\texttt{call}(\texttt{login}(\overline{uid})) \ \rhd\ \texttt{addLog}(uid); \\
\qquad\quad (\mu a_2.\ \hat{}\texttt{call}(\texttt{logout}()) \ \rhd\ \texttt{skip};\ a_1 \\
\qquad\qquad \square\ \hat{}\texttt{call}(\texttt{read}(\overline{f})) \ \rhd\ \texttt{addLog}(f);\ a_2 \\
\qquad\qquad \square\ \hat{}\texttt{call}(\texttt{login}(x)) \rhd\ \texttt{abort}();\ a_2) \\
\qquad \square\ \hat{}\texttt{call}(\texttt{logout}()) \ \rhd\ \texttt{abort}();\ a_1
\end{array}
$$

After simplifications, this resulting aspect is conflict free. In this derivation, apart from the laws of Figure 2, we have also used the law: $(\texttt{abort}() \bowtie I) = (I \bowtie \texttt{abort}()) = \texttt{abort}()$.

Note that the requirements of a composed aspect $A_1 \| A_2$ are determined by the parallel composition of the requirements of both constituent aspects. In our example, since *Flat* is required by both aspects and $Flat \| Flat = Flat$, it is also a requirement for $(SULog \| Log)$.

When an aspect is to be woven with a specific base program, it is necessary to check that the base program satisfies the aspect requirements. We now detail how this check can be done. We assume that the result of a control flow analysis (CFA from here on) of the base program[†] is expressed using the following grammar of regular expressions:

$$
\begin{array}{lll}
J & ::= & \mathtt{f} J_1\ \ldots\ J_n \ \mid\ ? \qquad \textit{; abstract join point} \\
S & ::= & \mu s.S \qquad \textit{; rec. def. } (s \in \mathcal{R}ec) \\
 & \mid & (J_1 \rightarrow S_1) [\![\ldots [\![(J_n \rightarrow S_n) \qquad \textit{; union of sequences} \\
 & \mid & s \qquad \textit{; end of recursion}
\end{array}
$$

[†]Note that the (possibly expensive OO features, *e.g.,* , taking into account nested calls) CFA of the base program must be performed only once and can be reused for different aspects.

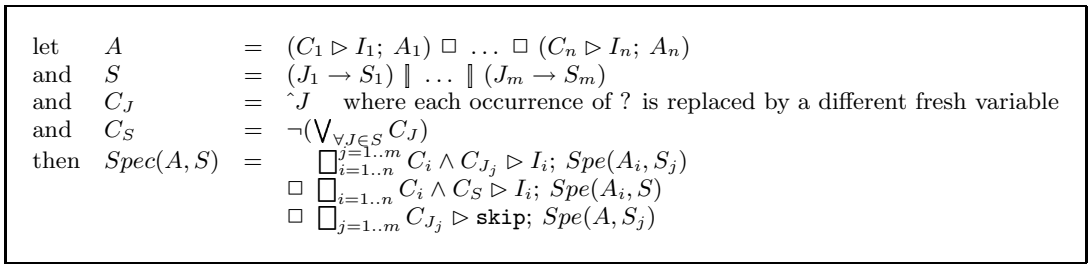

Figure 4: Aspect specialization *w.r.t.* a base program abstraction

The result of a CFA denotes sequences (traces) of (abstract) join points J. An abstract join point is a term with unknown values noted '?'. A sequence S is either a recursive definition, or a union of several execution sequences (each sequence starts with a join point). For instance, the traces of a base program performing sequences of flat sessions could be abstracted as follows:

$$Base = \mu s.\texttt{call}(\texttt{login}(?)) \to \texttt{call}(\texttt{logout}()) \to s$$

An instrumented aspect can be specialized for such a context. The specialization algorithm is mainly based on the law shown in Figure 4. The algorithm terminates due to the regular nature of the language of sequences. There are three cases. First, both the aspect and the abstract base program evolve when the current aspect matches the current join point. Second, only the aspect evolves when the current crosscut definition cannot match any joint point of the abstract base program (*i.e.,* when the abstract base program cannot be crosscut by the current rule of the aspect). Third, only the abstract base program evolves when the current join point is not relevant for the aspect.

Once an aspect has been specialized and simplified *w.r.t.* a base program, the result may or may not contain the special insert `abort()` (which is part of the requirements). If this insert does not occur anymore, the base program satisfies the aspect requirements and the specialized aspect can be woven. If it still occurs, the base program may not satisfy the aspect requirements. In this last case, the user could either use another aspect, or modify the base program so that it satisfies the aspect requirements. Note that static CFAs yield safe approximations of the dynamic behavior; base programs that dynamically satisfy the requirement may thus seem unsuitable. So, another option would be to weave the specialized aspect (with occurrences of `abort()`). The aspect will then perform dynamic checks in order to detect actual violations of the requirements.

As an example, specialization of the instrumented log aspect *FlatLog w.r.t.* context *Base* yields:

$$\begin{array}{l} Spec(FlatLog, Base) = \\ \quad \mu a_1.\ \hat{}\texttt{call}(\texttt{login}(\overline{uid}))\ \rhd\ \texttt{addLog}(uid); \\ \qquad \mu a_2.\ \hat{}\texttt{call}(\texttt{logout}())\ \rhd\ \texttt{skip};\ a_1 \\ \qquad\quad \Box\ \hat{}\texttt{call}(\texttt{read}(\overline{f}))\ \rhd\ \texttt{addLog}(f);\ a_2 \end{array}$$

The insert `abort()` does not occur in the specialized aspect. The aspect *Log* can be woven with the program analyzed as *Base*, because it satisfies the aspect requirements.

In software engineering, explicit hypotheses, such as pre- and post-conditions in programming by contract, or dependencies in module systems, support software reuse. The explicit requirements we propose should similarly support reuse of aspects. However, reusable aspects should be based on abstract concepts at the design level. For instance, sessions could be described at the design level in terms of session beginning (*i.e.,* `login`), session end (*i.e.,* `logout`) and user identity (*e.g., uid*). Once a base program is provided, design-level concepts should be translated into implementation-level notions. Session-related design-level concepts could be translated to concrete method calls available on the implementation level, such as `openSession()` and `disconnect()` executed in the context of an instance of the class `User`. Such a correspondence could be expressed by specifications such as:

$\texttt{login}(\overline{uid})$ `is implemented by` $\overline{uid}$`.openSession()`
$\texttt{logout}(uid)$ `is implemented by Server.disconnect(`uid`)`

which perform the corresponding simple substitutions in the crosscut definitions of aspects at weaving time.

Use of the history of execution events as a basic mechanism for the definition of aspects has been proposed independently by several researchers, in particular Filman [8], Walker et al. [18], as well as the authors [2, 6, 5].

As to the formalization of aspects and weavers, different approaches have been advocated. Wand *et al.* propose a denotational semantics for a subset of AspectJ [19]. Lämmel formalizes method-call interception using a big-step semantics [12]. Douence *et al.* [6] model crosscut definitions with execution trace parsers and weavers with execution monitors. De Volder *et al.* [16] propose a meta-programming framework based on Prolog where crosscuts are specified by predicates on abstract syntax trees. Walker *et al.* [17] introduce an abstract machine to define the operational semantics of ML extended with aspects; Tucker and Krishnamurthi [15] rely on abstract machines as well. Andrews [1] models AOP by means of algebraic processes. In the tradition of process calculi, Jagadeesan *et al.* [9] propose a calculus of AOP where aspects are primitive abstractions.

Such models are a prerequisite to formally study properties such as aspect interactions. However, despite its importance, very few work has previously been done on aspect interaction and conflict resolution. Douence *et al.* [6] present an approach for manual proofs of independence. Sereni *et al.* [13] generalize AspectJ's `cflow` using regular expressions on the call stack. They focus on optimization but they point out that their technique could also be used to detect interactions. Finally, interaction issues also arise in closely related fields of software engineering. For instance, Sihman *et al.* [14] use model checking to detect superimposition interactions and a large body of work is devoted to feature interactions (*e.g.,* Felty *et al.* [7]).

Concerning reuse, aspects are often advocated as reusable

pieces of software. It is true that AOP can sometimes avoid duplicating code. However, in order to make them fully reusable, module and software composition techniques should be adapted to aspects. Kienzle *et al.* [11] represent properties of aspects (namely, whether they provide, require and remove services) with a graph. An aspect can be reused in a configuration when it can be inserted into the corresponding dependencies graph. Sihman *et al.* [14] modularize proof obligations for superimpositions and perform checks before a superimposition is applied (*i.e.,* when an aspect is woven).

In this article, we have extended our generic formal framework for stateful aspects in three directions. The introduction of variables improves the expressive power of the framework and makes it possible to define more precise aspects. The main challenge was to design such an extension while retaining static interaction analysis capabilities. We have proposed a composition language built upon the same base as aspects. It is very general and can take into account the history of computation. Composition adaptors provide expressive means to deal with conflicts among interacting aspects. We have shown that the composition operators introduced in [4] can now be easily defined using this general composition language. Requirements were also defined using the same operators as stateful aspects. They address reusability by making explicit the validity domain of aspects. This extension makes interaction analysis more precise because requirements rule out some spurious interactions. Requirements can be seen as providing a pragmatic interaction analysis lying between strong independence (which can be too strong a condition) and weak independence [4] (which can be too costly).

[1] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proc. of the 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection)*, LNCS, pages 187–209. Springer Verlag, 2001.

[2] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 54–66, N.Y., Jan. 19–21 2000. ACM Press.

[3] H. Comon. Disunification: A survey. In *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, Cambridge, MA, 1991.

[4] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of the Conf. on Generative Programming and Component Engineering*, pages 173–188, 2002.

[5] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In M. Akşit et al., editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. to appear.

[6] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection)*, volume 2192 of *LNCS*, pages 170–186. Springer Verlag, 2001.

[7] A. P. Felty and K. S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):3–27, 2003.

[8] R. E. Filman and K. Havelund. Realizing aspects by transforming for events. In IEEE, editor, *Automated Software Engineering (ASE)*, Sept. 2002.

[9] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. of ECOOP 2003*, pages 415–427, 2003.

[10] G. Kiczales et al. Aspect-oriented programming. In *Proc. of ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[11] J. Kienzle, Y. Yu, and J. Xiong. On composition and reuse of aspects. In *Proc. of the 2nd Foundations of Aspect-oriented Languages Workshop at AOSD 2003*, pages 17–24, 2003.

[12] R. Lämmel. A semantics for method-call interception. In *1st Int. Conf. on Aspect-Oriented Software Development (AOSD'02)*, 2002.

[13] D. Sereni and O. de Moor. Static analysis of aspects. In *Proc. of the 2nd Int. Conf. on Aspect-oriented Software Development*, pages 30–39. ACM Press, 2003.

[14] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, 2003.

[15] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD)*, pages 158–167. ACM Press, 2003.

[16] K. D. Volder. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[17] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proc. of the Int. Conf. on Functional Programming*, 2003.

[18] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. Proc. Int. WS on Advanced Separation of Concerns at ICSE, 2001.

[19] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *FOOL 9*, pages 67–88, 2002.