Aspects Preserving Properties

Simplice Djoko Djoko, Rémi Douence, Pascal Fradet

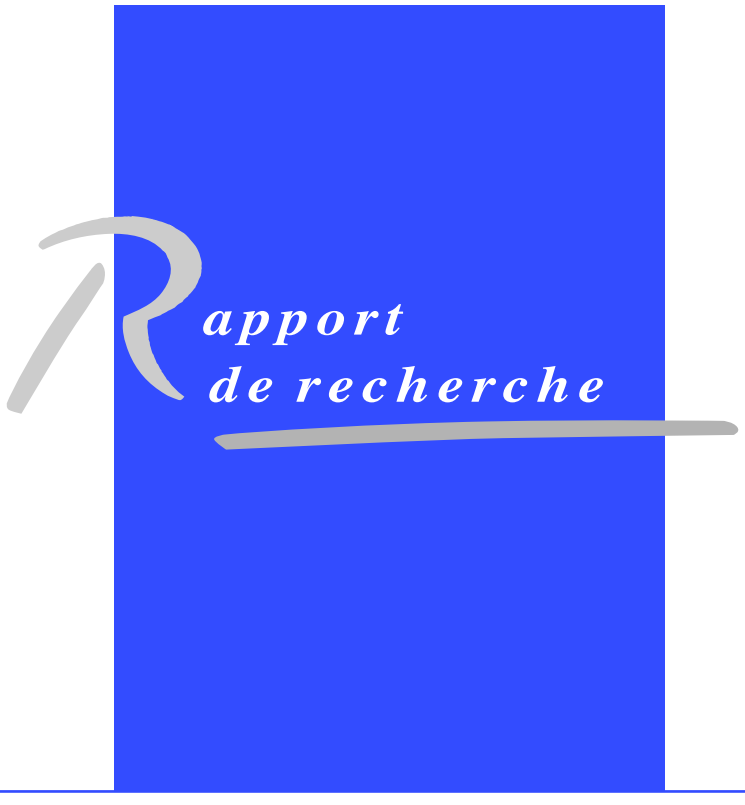
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Aspects Preserving Properties

Simplice Djoko Djoko — Rémi Douence — Pascal Fradet

N° 7155

December 2009



ISSN 0249-6399 ISRN INRIA/RR--7155--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Aspects Preserving Properties

Simplice Djoko Djoko*† , Rémi Douence‡§, Pascal Fradet¶ †

Thème : Systèmes embarqués et temps réel
Équipes-Projets PopArt et Ascola

Rapport de recherche n° 7155 — December 2009 — 49 pages

Abstract: Aspect Oriented Programming can arbitrarily distort the semantics of programs. In particular, weaving can invalidate crucial safety and liveness properties of the base program. In this article, we identify categories of aspects that preserve some classes of properties. Specialized aspect languages can be then designed to ensure that aspects belong to a specific category and therefore that woven programs will preserve the corresponding properties.

Our categories of aspects, inspired by Katz's, comprise observers, aborters and confiners. Observers introduce new instructions and a new local state but they do not modify the base program's state and control-flow. Aborters are observers which may also abort executions. Confiners only ensure that executions remain in the reachable states of the base program. These categories (along with three other) are defined precisely based on a language independent abstract semantics framework. The classes of preserved properties are defined as subsets of LTL for deterministic programs and CTL* for non-deterministic ones. We can formally prove that, for any program, the weaving of any aspect in a category preserves any property in the related class.

We present, for most aspect categories, a specialized aspect language which ensures that any aspect written in that language belongs to the corresponding category. It can be proved that these languages preserve the corresponding classes of properties by construction. The aspect languages share the same expressive pointcut language and are designed *w.r.t.* a common imperative base language.

Each category and language are illustrated by simple examples. The appendix provides semantics and examples of proofs: the proof of preservation of properties by a category and the proof that all aspects written in a language belong to the corresponding category.

Key-words: Aspect weaving, proofs, semantics, temporal properties

* INRIA, EMN, LINA
† Inria Grenoble - Rhône-Alpes, 655, av. de l'Europe, 38330 Montbonnot, France
‡ EMN, INRIA, LINA
§ École des Mines de Nantes, 4, rue Alfred Kastler, 44307 Nantes Cedex 3, France
¶ INRIA

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Aspects et préservation de propriétés

Résumé : Le tissage d'aspects peut bouleverser la sémantique du programme de base. En particulier, le tissage d'un aspect peut faire perdre des propriétés de sûreté ou vivacité clés de l'application. Dans cet article, nous identifions des catégories d'aspects qui préservent certaines classes de propriétés. Nous proposons des langages d'aspects dédiés qui assurent que leurs aspects appartiennent à une catégorie donnée.

Nos catégories d'aspects, inspirées de celles de Katz, comprennent les observateurs, les terminateurs et les vérouilleurs. Les observateurs ajoutent de nouvelles instructions et états mais ne modifient pas le flot de contrôle ou les états du programme de base. Les terminateurs sont des observateurs qui peuvent de plus arrêter le programme de base. Les vérouilleurs garantissent seulement que les états du programme tissé restent dans l'ensemble des états accessible du programme de base. Ces catégories (ainsi que trois autres) sont définies précisement dans un cadre sémantique indépendent du langage de base. Les classes de propriétés préservées sont définies comme des sous-ensemble de LTL (pour les programmes déterministes) et CTL* (pour les programmes non déterministes). On peut montrer formellement que, pour tout programme, le tissage de tout aspect d'une catégorie donnée préserve toute propriété de la classe correspondante.

Nous présentons, pour la plupart des catégories, un langage spécialisé d'aspects qui assure que tout aspect écrit appartient à la catégorie correspondante. On peut montrer que ces langages assurent la préservation des classes de propriétés par construction. Ces langages d'aspects partagent le même langage de point de coupure et s'appliquent au même langage de base impératif.

Les catégories et langages sont illustrés par des exemples simples. L'appendice détaille la sémantique du langage de base et deux exemples de preuves: la preuve de préservation des propriétés pour une catégorie et la preuve que tous les aspects écrits dans un langage appartiennent à la catégorie correspondante.

Mots-clés : Aspects, tissage, preuves, sémantique, propriétés temporelles

1 Introduction

Aspect oriented programming (AOP) proposes to modularize concerns that crosscut the base program [26]. However, aspects can in general distort the semantics of the base program. The programmer may have to inspect the woven program (or to debug its execution) to understand its semantics. In this article, we consider several categories of aspects that alter the semantics of the base program in a tightly controlled manner. For each category of aspects $\mathcal{A}_x$, we identify a corresponding class of properties φ^x that is preserved by weaving these aspects. In other words, let P be a program that satisfies a property $\varphi \in \varphi^x$, then weaving any aspect $A \in \mathcal{A}_x$ on P will produce a program satisfying φ. Our categories of aspects, inspired by Katz's [24], comprise observers, aborters, confiners and weak intruders.

- *Observers* do not modify the base program's state and control-flow. Advice may only modify the aspect's local variables.

- *Aborters* are observers which may also abort executions. The program's state is not modified but its control flow may be terminated.

- *Confiners* may modify the state and control-flow but ensure that states remain in the reachable states of the base program.

- *Weak intruders* may modify states and control-flow with no restriction within the advice code. However, the execution of the base program code must involve only states already reachable by the unwoven program.

Typically, persistence, debugging, tracing, logging and profiling aspects are observers whereas aspects ensuring safety properties such as security aspects are aborters. Some optimization aspects (which may use shortcuts to reach future states) or fault-tolerance aspects (which roll-back to past states) may belong to the last two categories.

An observer can only insert advice which will write its own local variables. Intuitively, it should preserve many properties but caution must be exercised. For example, properties involving the absence of unwanted events (such as specific method calls) are often not preserved since the advice inserts new events. Liveness properties may also be violated if the advice fails to terminate. Further, we must ensure that base programs are not reflexive otherwise the base program control-flow could be indirectly modified by the most harmless looking advice. These examples should make it clear that such a taxonomy asks for a formal treatment.

We define the categories precisely based on a language independent abstract semantics framework. The classes of properties are defined as subsets of LTL [31] for deterministic programs and CTL* [3] for non deterministic ones. We can formally prove that, for any program, the weaving of any aspect in a category preserves any property in the related class.

To put these results into practice, we need to be able to determine whether an aspect belongs to a category. This process can rely on static analyses (*a posteriori* approach) or on specialized aspect languages (*a priori* approach). We choose the latter approach and present for each aspect category a restricted

aspect language which ensures that any aspect written in that language belongs to the corresponding category. Therefore, these languages ensure that the corresponding properties are preserved by construction. The aspect languages are designed for a simple imperative base language and use an expressive pointcut language. Each aspect language is illustrated using simple examples of aspects.

Section 2 introduces the formal framework used in the rest of the paper. It presents in particular our common aspect semantics base (Section 2.1), the base and woven execution traces (Section 2.2), and the properties based on temporal logic (Section 2.3).

We define in Section 3 the categories of aspects and their corresponding classes of temporal properties: observers (Section 3.1.1), aborters (Section 3.1.2), confiners (Section 3.1.3) and weak intruders (Section 3.1.4). Non determinism suggests two new categories of aspects: selectors (Section 3.2.1) and regulators(Section 3.2.1). Our presentation of aspect categories concludes by a study of composition and interactions between the different kinds of aspects (Section 3.3).

Section 4 introduces the imperative (base and advice) language (Section 4.1), its associated pointcut language (Section 4.2) and several aspect languages corresponding to observers (Section 4.3.1), aborters (Section 4.3.2) and confiners (Section 4.3.3) for a deterministic setting, and selectors (Section 4.4.2) and regulators (Section 4.4.3) for non deterministic languages.

Section 5 reviews some related work and Section 6 discusses possible future research directions and concludes. The appendix provides the semantics of the base language and two examples of proofs: the preservation of properties by observers and the proof that all aspects written in the observer language are indeed observers.

This article combines, revises and extends two conference papers presented in PEPM'08 [12] and SEFM'08 [10]. It is also based on a French PhD thesis [11].

2 Framework

In order to prove that properties are preserved by weaving, we have to define the semantics of base and woven programs. We do so using a Common Aspect Semantics Base (CASB) for AOP [14]. That abstract framework applies to most base and aspect languages. We define execution traces of base and woven programs and we show how they are related. We then recall the main characteristics of linear and branching temporal logic used to express properties on deterministic and non deterministic programs respectively.

2.1 The Common Aspect Semantics Base

The CASB relies on the small step semantics of the base language which is supposed to represent the semantics of advice as well. That semantics is described through a binary relation $\rightarrow_b$ on configurations (C, Σ) made of a program and a state:

- a program C is a sequence of basic instructions i terminated by $\bullet$:

$$C ::= i : C \mid \bullet$$

- states Σ are kept as abstract as possible. They may contain environments (*e.g.,* associating variables to values, procedure names to code, etc.), stacks (*e.g.,* evaluation stack), heaps (*e.g.,* dynamically allocated memory), etc.

A single reduction step of the base language semantics is written

$$(i : C, \Sigma) \rightarrow_b (C', \Sigma')$$

Intuitively, i represents the current instruction and C the continuation. The component $i : C$ can be seen as a control stack. The operator ":" sequences the execution of instructions. The semantics of the base language used in Section 4.1 is expressed along those lines (see appendix B). The interested reader will also find in [14] the semantic description of a core Java language (Featherweight Java with assignments) in that form.

In the following, woven configurations (C, Σ) are supposed to be made of the following components:

- C is the sequence of instructions of the woven program. We write i_b for a base program instruction and i_a for an advice instruction. The instruction ϵ, which represents the final instruction of a program, is considered as an i_b instruction;

- Σ^b is the subset of the state Σ corresponding to the state of the base program (*i.e.,* the variables, environment, heap, modified by i_b instructions and possibly by i_a instructions);

- Σ^a is the subset of Σ that corresponds to the local state of aspects (*i.e.,* the variables, environment, heap, *etc.* which cannot be modified by i_b but only i_a instructions);

- Σ^ψ is the subset of Σ that represents aspects. It is a function that decides whether the current instruction should be woven and transforms the configuration accordingly. When a new instance of an aspect is created, both Σ^a and Σ^ψ are modified.

Let (C, Σ) be a woven configuration then $\Sigma = \Sigma^b \cup \Sigma^a \cup \Sigma^\psi$. Reduction of woven programs has the following properties:

$$\forall (C, \Sigma).(i_b : C, \Sigma) \rightarrow_b (C', \Sigma') \text{ with } \Sigma' = \Sigma'^b \cup \Sigma^a \cup \Sigma^\psi$$

that is, the reduction of a base program instruction can only modify the state of the base program, and

$$\forall (C, \Sigma).(i_a : C, \Sigma) \rightarrow_b (C', \Sigma') \text{ with } \Sigma' = \Sigma'^b \cup \Sigma'^a \cup \Sigma^\psi$$

that is, the reduction of an advice instruction can, in general, modify both the state of the base program and the local state of aspects.

The semantics of woven reduction is represented by the binary relation $\rightarrow$ defined by:

$$\textsc{Reduce} \quad \frac{(C, \Sigma) \rightarrow_b (C', \Sigma') \qquad w(C', \Sigma') = (C'', \Sigma'')}{(C, \Sigma) \rightarrow (C'', \Sigma'')}$$

A reduction step $\rightarrow$ of the woven program first reduces the first instruction of the current configuration using $\rightarrow_b$, then weaves the reduced configuration using the function w. The weaving function w is defined by two rules:

- either, the current instruction is not matched by the aspects (Σ^ψ returns nil) and w returns the configuration unchanged

$$\textsc{weave0}\quad \frac{\Sigma^\psi(C,\Sigma) = nil}{w(C,\Sigma) = (C,\Sigma)}$$

- or the current instruction is matched by the aspects and Σ^ψ returns a new configuration (C',Σ')

$$\textsc{weave1}\quad \frac{\Sigma^\psi(C,\Sigma) = (C',\Sigma') \qquad w(C',\Sigma') = (C'',\Sigma'')}{w(C,\Sigma) = (C'',\Sigma'')}$$

  where

  - C' is the new code in which an advice is inserted before, after or around the current instruction of C (see [14] for more details);
  - $\Sigma' = \Sigma^b \cup \Sigma'^a \cup \Sigma'^\psi$, with Σ'^ψ which may contain a new aspect instance and Σ'^a its corresponding new state.

Note that weaving can be recursively applied on the code of a newly introduced advice. In some cases, we should prevent some instructions to be matched. For example, an aspect matching an instruction i and inserting a before advice a should not match i again just after executing a. We used tagged instructions such as $\bar{i}$ which have exactly the same semantics as i except that it is not subject to weaving. Formally

$$\textsc{Tagged}\quad \frac{(i : C,\Sigma) \to_b (C',\Sigma')}{(\bar{i} : C,\Sigma) \to (C',\Sigma')}$$

We assume that weaving only depends on the current instruction (not on the continuation). The interested reader will find in [14] the semantics of common aspectual features in that framework (*e.g.,* before, after and around aspects, cflow pointcuts, aspects on exceptions, aspect deployment, aspect instantiation, etc.).

Since weaving is always performed after a $\to_b$ reduction, it is not possible to weave the first instruction. In some cases, it might be useful to start the program by a before-advice. In order to allow such weaving, we introduce a special instruction *start* and we assume that initial configurations are of the form $(start : C,\Sigma)$. The semantics of *start* is the same as a *nop* (no operation):

$$(start : C,\Sigma) \to_b (C,\Sigma)$$

So, a base program always starts by the reduction step

$$(start : C_0,\Sigma_0) \to_b (C_0,\Sigma_0)$$

whereas a woven execution starts by the reduction step

$$(start : C_0,\Sigma_0) \to (C'_0,\Sigma'_0) \qquad \text{with} \quad w(C_0,\Sigma_0) = (C'_0,\Sigma'_0)$$

2.2 Base and Woven Execution Traces

In the following, programs are represented by their execution traces. Terminating programs ends by a final instruction ϵ and final configurations are of the form $(\epsilon : \bullet, \Sigma)$. For simplicity and regularity, we only consider infinite traces. In order to do so, the final instruction ϵ is supposed to have the following reduction rule:

$$\forall \Sigma.(\epsilon : \bullet, \Sigma) \rightarrow_b (\epsilon : \bullet, \Sigma)$$

This way, non-terminating and terminating programs will be both represented as infinite execution traces.

The base program execution trace, with (C_0, Σ_0) as initial configuration, will be denoted by $\mathcal{B}(C_0, \Sigma_0)$ (definition 2.1).

Definition 2.1.

$$\begin{array}{rl} \mathcal{B}(C_0, \Sigma_0) = & (i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots \\ \textit{with} & \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow_b (i_{j+1} : C_{j+1}, \Sigma_{j+1}) \end{array}$$

We write $\mathcal{W}(C_0, \Sigma_0)$ for the infinite woven execution trace (definition 2.2).

Definition 2.2.

$$\begin{array}{rl} \mathcal{W}(C_0, \Sigma_0) = & (i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots \\ \textit{with} & \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow (i_{j+1} : C_{j+1}, \Sigma_{j+1}) \end{array}$$

Since traces are used to define properties which concern only states and current instructions, the continuation (the control stack) does not appear in traces. Note that in both definitions, the initial instruction i_0 (*i.e., start*) does not appear.

The semantics of non-deterministic programs is defined as sets of (infinite) execution traces. The categories of aspects are defined based on this semantics and the same auxiliary functions ($proj_b$ and $preserve_b$). We abstract the base and woven program executions as sets of infinite traces written $\mathcal{B}^*(C_0, \Sigma_0)$ (Definition 2.3) and $\mathcal{W}^*(C_0, \Sigma_0)$ (Definition 2.4).

Definition 2.3.

$$\mathcal{B}^*(C_0, \Sigma_0) = \{(i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots \mid \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow_b (i_{j+1} : C_{j+1}, \Sigma_{j+1})\}$$

Definition 2.4.

$$\mathcal{W}^*(C_0, \Sigma_0) = \{(i_1, \Sigma_1) : (i_2, \Sigma_2) : \ldots \mid \forall (j \geq 0).(i_j : C_j, \Sigma_j) \rightarrow (i_{j+1} : C_{j+1}, \Sigma_{j+1})\}$$

In the rest of the paper, if α is a trace then its i^{th} element is denoted by α_i and prefix, postfix and subtraces are written as follows:

$$\begin{array}{rcl} \alpha_{\rightarrow j} & = & \alpha_1 : \ldots : \alpha_j \\ \alpha_{j \rightarrow} & = & \alpha_j : \alpha_{j+1} \ldots \\ \alpha_{i \rightarrow j} & = & \alpha_i : \ldots : \alpha_j \end{array}$$

with $i > 0$ and $j > 0$. The empty trace can be written $\alpha_{\rightarrow 0}$.

The relations between the base and woven execution traces is expressed using the functions $proj_b$ and $preserve_b$. We write $Traces_{\mathcal{B}}$, $Traces_{\mathcal{W}}$ and $Sequence_{i_b}$

to denote the sets of base program execution traces, woven execution traces and sequences of base instructions respectively.

The function $proj_b$ projects a base or woven trace on the sequence of the base instructions which have been executed.

$$\begin{array}{l} proj_b : Traces_{\mathcal{B}} \cup Traces_{\mathcal{W}} \to Sequence_{i_b} \\ proj_b((i_b, \Sigma) : T) \;=\; i_b : (proj_b\ T) \\ proj_b((i_a, \Sigma) : T) \;=\; proj_b\ T \end{array}$$

The predicate $preserve_b$ checks that the advice instructions in a woven trace do not modify Σ^b. Each i_a instruction must leave the state of the base program (Σ^b) unchanged.

$$\begin{array}{l} preserve_b : Traces_{\mathcal{W}} \to bool \\ preserve_b(\tilde{\alpha}) \;=\; \forall (j \geq 1).\ \tilde{\alpha}_j = (i_a, \Sigma_j) \;\Rightarrow\; \tilde{\alpha}_{j+1} = (i, \Sigma_{j+1}) \;\wedge\; \Sigma^b_j = \Sigma^b_{j+1} \end{array}$$

These functions are used to define aspect categories.

2.3 Properties

Temporal logic permits to define a wide range of properties of program executions [31]. Security properties or more generally, invariant, liveness or safety properties are naturally expressed in temporal logic.

Temporal properties are defined over execution traces. We start by defining the atomic propositions considered in this article. We define the syntax and semantics of LTL formulae *w.r.t.* our (base and woven) execution traces. We review standard classes of LTL properties and briefly discuss why these classes are not, in general, preserved by weaving. We conclude by presenting along the same lines the branching temporal logic CTL* that we use to express properties of non-deterministic programs.

2.3.1 Atomic propositions

In our context, an atomic proposition ap of LTL is either an atomic proposition sp on states Σ (*e.g.*, $x \geq 0$ which is *true* when the variable x is positive is the current state), or an atomic proposition ep on instructions or events (*e.g.*, `foo` which is *true* when the current instruction is a call to method `foo`).

An atomic proposition ap is *true* at a step of a (base or woven) trace α_j iff α_j satisfies ap denoted by $\alpha_j \models ap$. This is defined based on the two following auxiliary functions:

- The function $m :: Instruction \times Ep \to bool$, where $Instruction$ is the set of instructions and Ep the set of atomic propositions on instructions, returns *true* if the proposition matches the current instruction. The function m is overloaded in order to take a trace element as parameter:

$$\begin{array}{lcl} m :: Step \times Ep & \to & bool \\ m((i, \Sigma), ep) & = & m(i, ep) \end{array}$$

- The function $l :: State_B \times Sp \rightarrow bool$, where $State_B$ is the set of Σ^b and Sp the set of atomic propositions on Σ^b ($Sp \subset Ap$), returns *true* if the proposition is satisfied by the state passed as parameter. The function l is overloaded in order to take a trace element as parameter:

$$\begin{array}{lcl} l :: Step \times Sp & \rightarrow & bool \\ l((i,\Sigma), sp) & = & l(\Sigma^b, sp) \end{array}$$

Then, $\alpha_j \models ap$ is defined as follows:

$$\begin{array}{lcl} \alpha_j \models ep & \Leftrightarrow & m(\alpha_j, ep) = true \\ \alpha_j \models \neg ep & \Leftrightarrow & m(\alpha_j, ep) = false \\ \alpha_j \models sp & \Leftrightarrow & l(\alpha_j, sp) = true \\ \alpha_j \models \neg sp & \Leftrightarrow & l(\alpha_j, sp) = false \end{array}$$

2.3.2 Semantics of LTL

We consider LTL formulae in positive normal form *i.e.,* where negation occurs only on atomic propositions (Grammar 2.5). In φ, the operator $\bigcirc$ is read "next", $\cup$ is read "until", and W is read "weak until".

Grammar 2.5.

$$\varphi ::= ap \mid \neg ap \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \cup \varphi_2 \mid \varphi_1 W \varphi_2$$

The semantics of an LTL formula is defined on a trace α as follows:

$$\begin{array}{lcl} \alpha \models ap & \Leftrightarrow & \alpha_1 \models ap \\ \alpha \models \neg ap & \Leftrightarrow & \alpha_1 \models \neg ap \\ \alpha \models \varphi_1 \vee \varphi_2 & \Leftrightarrow & \alpha \models \varphi_1 \vee \alpha \models \varphi_2 \\ \alpha \models \varphi_1 \wedge \varphi_2 & \Leftrightarrow & \alpha \models \varphi_1 \wedge \alpha \models \varphi_2 \\ \alpha \models \bigcirc \varphi & \Leftrightarrow & \alpha_{2\rightarrow} \models \varphi \\ \alpha \models \varphi_1 \cup \varphi_2 & \Leftrightarrow & \exists (j \geq 1). \alpha_{j\rightarrow} \models \varphi_2 \ \wedge \ \forall (1 \leq i < j). \alpha_{i\rightarrow} \models \varphi_1 \\ \alpha \models \varphi_1 W \varphi_2 & \Leftrightarrow & \forall (j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_1 \ \vee \ \alpha \models \varphi_1 \cup \varphi_2 \end{array}$$

The atomic proposition ap (resp. $\neg ap$) is *true* on α if ap is *true* (resp. *false*) on the first element of α; $\varphi_1 \vee \varphi_2$ is *true* if φ_1 is *true* or φ_2 is *true*; $\varphi_1 \wedge \varphi_2$ is *true* if φ_1 is *true* and φ_2 is *true*; $\bigcirc \varphi$ is *true* if φ is *true* on the trace immediately following; $\varphi_1 \cup \varphi_2$ is *true* if φ_1 is *true* until φ_2 becomes *true*; finally $\varphi_1 W \varphi_2$ is *true* if φ_1 is always *true* or $\varphi_1 \cup \varphi_2$ is *true*.

For the sake of readability, derived operators can be defined:

- $\Diamond \varphi = true \cup \varphi$ is read "eventually φ" *i.e.,* in the future, there is a (postfix) trace that satisfies φ;

- $\Box \varphi = \varphi \ W false$ is read "always φ" *i.e.,* all (postfix) traces in the trace satisfy φ.

2.3.3 Standard Classes of Temporal properties

Standard classes of temporal properties [36] comprise:

- liveness properties: "something (good) eventually happens". Liveness properties are of the form $\Diamond\varphi$. Liveness properties can also be repeated to express fairness (*i.e.,* "something eventually happens infinitely often"). In this case, they are of the form $\Box\Diamond\varphi$;

- safety properties: "something (bad) never happens". Safety properties are of the form $\Box\varphi$;

- Invariant properties: "something always happens". They are of the form $\Box\varphi$ where φ is composed of atomic propositions, negations, disjunctions and conjunctions but no temporal operators. They are a subset of safety properties which do not relate to the history of the computation.

These classes are very expressive since any LTL property can be expressed as a conjunction of a safety and a liveness property. In general, they are not preserved by aspect weaving. For instance, consider the liveness property $\Diamond$`backup` meaning that the `backup` procedure is eventually called (*i.e.,* "the state of the system is eventually saved"). An around aspect replacing calls to the function `backup` by different calls will violate the liveness property. Regarding safety properties, consider a base program that never calls the function `diskformat` and therefore satisfies the property $\Box\neg$`diskformat`. An aspect that calls this function in its advice will violate the property.

Section 3 is devoted to identifying categories of aspects that preserve large classes of temporal properties.

2.3.4 Branching temporal logic CTL*

In the non-deterministic case, classes of properties are subsets of the branching temporal logic CTL* [3]. Grammar 2.6 defines the positive normal form of CTL* formulae.

Grammar 2.6.

$$\theta \quad ::= \quad ap \;\mid \neg ap \;\mid \theta_1 \vee \theta_2 \;\mid \theta_1 \wedge \theta_2 \;\mid \exists\omega \;\mid \forall\omega$$

$$\omega \quad ::= \quad \theta \;\mid \omega_1 \vee \omega_2 \;\mid \omega_1 \wedge \omega_2 \;\mid \bigcirc\omega \;\mid \omega_1 \cup \omega_2 \;\mid \omega_1 W \omega_2$$

Whereas LTL specifies properties on an execution trace, CTL* specifies properties on a set of execution traces. CTL* extends LTL with the logical quantifiers $\exists\omega$ ("there exists traces satisfying ω") and $\forall\omega$ ("all traces satisfy ω"). It is strictly more expressive than LTL. Any LTL property p for a trace α is equivalent to the CTL* formula $\forall p$ for the set $\{\alpha\}$. In Grammar 2.6, θ represents properties on trace steps and ω properties on traces.

The semantics of CTL* is quite similar to the semantics of LTL defined above. The semantics of logical quantifiers is defined as follows:

$$\begin{array}{lcl} T, \alpha_j \models \exists\omega & \Leftrightarrow & \exists(\alpha \in T).T, \alpha \models \omega \\ T, \alpha_j \models \forall\omega & \Leftrightarrow & \forall(\alpha \in T).T, \alpha \models \omega \end{array}$$

In these definitions, the environment T is the set of traces starting from α_j. In our context, T will be initially either $\mathcal{B}^*(C_0, \Sigma_0)$ or $\mathcal{W}^*(C_0, \Sigma_0)$. A step α_j

satisfies $\exists\omega$ if there exists an execution $\alpha \in T$ (*i.e.*, traces from α_j) that satisfies ω. A step α_j satisfies $\forall\omega$ if all execution traces $\alpha \in T$ satisfy ω. The derived operators $\Diamond$ and $\Box$ are defined in CTL* in the same way as in LTL.

3 Aspects Categories

Our aspect categories comprise observers, aborters, confiners and weak intruders starting from the least to the most expressive/invasive. For each category $\mathcal{A}_x$, we present a class of properties φ^x (a subset of LTL) which are preserved by the weaving of any aspect of $\mathcal{A}_x$. Non determinism brings two new categories: selectors and regulators. The classes of preserved properties are in this case subsets of CTL*. We conclude the section by studying the composition (and the potential interaction) of aspects belonging of different categories.

3.1 Deterministic case

The four aspect categories observers ($\mathcal{A}_o$), aborters ($\mathcal{A}_a$), confiners ($\mathcal{A}_c$) and weak intruders ($\mathcal{A}_w$) are related by inclusion:

$$\mathcal{A}_o \subset \mathcal{A}_a \subset \mathcal{A}_c \subset \mathcal{A}_w$$

The observer category is the most restricted category; it is included in all the other. The weak intruder category is the most expressive category; it includes all the other. For instance, an aborter is also a confiner and a weak intruder. The corresponding classes of properties are also related by inclusion:

$$\varphi^o \supset \varphi^a \supset \varphi^c \supset \varphi^w$$

Not surprisingly, the most restricted category of aspects ($\mathcal{A}_o$) preserves the largest class of properties (φ^o) and the inclusion chain is in the opposite direction.

An important point to keep in mind is that our preservation proofs should stand for any program, any aspect of the category and any property of the class. Our course, for a specific program and aspect many more properties might be preserved. The advantage of this approach is when an aspect is shown to belong to a category, then we know a large class of properties that will be preserved whatever the program is. Preservation is robust *w.r.t.* program changes.

For these reasons, the classes of preserved properties cannot include the temporal operator $\bigcirc$. Indeed, a trace satisfies $\bigcirc\varphi$ only if the sequence immediately following satisfies φ. The weaving of even the most harmless aspect (for example, an aspect inserting a *nop* instruction) fails to preserve this kind of property. It suffices to weave it just before φ becomes satisfied. Since all aspects introduce extra steps in the execution trace, no category of aspects preserves $\bigcirc$-properties for all programs.

In the following, we explain our categories and classes using small examples of execution traces where only the relevant satisfied properties are shown. For example:

$$x = 0 : x = 0 : (x = 1, \mathit{print}) : \epsilon : \epsilon : \ldots$$

represents an execution trace where the first and the second steps satisfy $x = 0$ and the third step satisfies $x = 1$ and has *print* as its current instruction

(*e.g.*, the second instruction has changed the value of x). This trace satisfies, for example, the property $(x = 0)W print$.

3.1.1 Observers

An observer (Definition 3.1) does not modify the control-flow of the base program but only inserts advice instructions i_a. The woven and the base execution traces can be projected (using $proj_b$) onto the same sequence of base instructions. An observer does not modify the state of the base program: advice instructions i_a do not change the base state Σ^b. This is the property checked by the predicate $preserve_b$.

Definition 3.1.

$$\forall (C, \Sigma).\ \ \Sigma^\psi \in \mathcal{A}_o \ \Leftrightarrow\ proj_b(\alpha) = proj_b(\tilde{\alpha}) \ \wedge\ preserve_b(\tilde{\alpha})$$
$$with\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)$$

Definition 3.1 states that observers may only modify execution traces by inserting new advice instructions (i_a) and a new local state (Σ^a). Note that this definition also implies that the advice terminates.

The class of properties φ^o preserved by observer aspects are defined by the grammar 3.2.

Grammar 3.2.

$$\varphi^o \ ::=\ sp \mid \neg sp \mid \varphi_1^o \vee \varphi_2^o \mid \varphi_1^o \wedge \varphi_2^o \mid \varphi_1^o \cup \varphi_2^o \mid \varphi_1^o W \varphi_2^o \mid true \cup \varphi'^o$$

$$\varphi'^o \ ::=\ ep \mid \neg ep \mid sp \mid \neg sp \mid \varphi_1'^o \vee \varphi_2'^o \mid \varphi_1'^o \wedge \varphi_2'^o \mid \varphi_1^o \cup \varphi_2^o \mid \varphi_1^o W \varphi_2^o \mid true \cup \varphi'^o$$

As in the previous section, the variables sp and ep refer to atomic propositions on the base state and instructions respectively. The language φ^o is LTL without the $\bigcirc$ operator when atomic propositions are state propositions (sp). So, it can express all safety, liveness and invariant properties (without $\bigcirc$) on base states Σ^b. The class is more restricted when the property involves atomic propositions on events (ep). These properties can only occur as $true \cup \varphi'^o$. This makes it possible to define liveness properties on events. Indeed, a liveness property $\Diamond \varphi'^o$ can be rewritten as $true \cup \varphi'^o$ and a liveness fair property $\Box\Diamond \varphi'^o$ can be rewritten as $(true \cup \varphi'^o) W false$. On the other hand, this language forbids safety properties on events. A safety property $\Box \neg \varphi$ is of the form $(\neg \varphi) W false$ which does not belong to grammar 3.2. Intuitively, safety properties on events forbid some sequences of instructions. An observer introduces sequences of instructions, so it may introduce a forbidden sequence of instructions in particular. For example, the base program sequence

$$x = 0 : x = 0 : (x = 1, print) : \epsilon : \epsilon : \ldots$$

satisfies $(x = 0) \cup print$ and $(x = 0) W print$, but after the weaving of the advice instruction $write$ just before $print$

$$x = 0 : x = 0 : (x = 1, write) : (x = 1, print) : \epsilon : \epsilon : \ldots$$

both properties are not satisfied any more. Also, the property *readWfalse* (*i.e.,* always *read*) is satisfied by the infinite trace of *read* instructions

$$read : read : read : \ldots$$

but after the weaving of the advice *write* after the first *read*

$$read : write : read : read : \ldots$$

the property is not satisfied any more.

The theorem 3.3 formally states that the weaving of an observer preserves all properties in φ^o which were satisfied by the base program. The appendix presents the proof of this theorem.

Theorem 3.3.

$$\forall(C,\Sigma).\ \ \Sigma^{\psi} \in \mathcal{A}_o \ \Rightarrow\ \forall(p \in \varphi^o).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p$$
$$with\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ \ and\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Persistence, debugging, tracing, logging and profiling aspects typically belong to the class of observers. Persistence aspects which only store the states of the base program during its execution on a data base are clearly observers. Debugging aspects printing variables of the base program or inserting breakpoints are observers. However, a debugger aspect allowing the user to interactively change the base program state would fail to be an observer. Tracing, logging or profiling aspects usually only observe the execution of the base program and write information on this execution (*e.g.,* method calls, parameters values, etc.) in a file. An example of profiling aspects is runtime analysis aspects such as intrusion detection aspects which observe the execution, detect suspicious behaviors and warn administrators.

In the documentation of AspectJ, there are many profiling aspects such as `telecom/TimerLog`, `tracing/lib/TraceMyClasses`, `tjp/GetInfo`, ... In [2], Govidranj *et al.* present a tool named InfraRED. It is based on several observer AspectJ aspects to monitor J2EE applications and to detect and analyze performance problems.

3.1.2 Aborters

An aborter (Definition 3.4) does not modify the state of the base program. As in the previous definition of observers, the predicate $preserve_b$ holds for the woven trace. However, an aborter can modify the control-flow by terminating the execution of the woven program. This is modeled by an i_a instruction `abort` which reduces any configuration into the final one:

$$\forall(C,\Sigma).\ (\texttt{abort} : C,\Sigma) \rightarrow (\epsilon : \bullet,\Sigma)$$

If `abort` is never executed, the projections of the base and woven traces are equal; the aborter behaves like an observer. The projection of an aborted woven trace on i_b is a prefix of the projection of the base program trace. After this point, all instructions are equal to ϵ.

Definition 3.4.

$$\begin{array}{lll} \forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_a & \Leftrightarrow & preserve_b(\tilde{\alpha}) \ \wedge \ proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ & \vee & \exists(i \geq 0).\exists(j \geq i).\ proj_b(\alpha_{\rightarrow i}) = proj_b(\tilde{\alpha}_{\rightarrow j}) \\ & & \qquad \wedge \ \forall(k > j).\tilde{\alpha}_k = (\epsilon, \_) \\ with \ \ \alpha = \mathcal{B}(C,\Sigma) & and & \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Note that this definition rules out aspects whose advice does not terminate ($proj_b(\alpha) = proj_b(\tilde{\alpha}) \ \vee \ \forall(k > j).\tilde{\alpha}_k = (\epsilon, \_)$).

Observers are included in the category of aborters. The set of properties preserved by aborters (Grammar 3.5) is a subset of the set of properties preserved by observers (Grammar 3.2).

Grammar 3.5.

$$\varphi^a ::= sp \mid \neg sp \mid \varphi_1^a \vee \varphi_2^a \mid \varphi_1^a \wedge \varphi_2^a \mid \varphi_1^a W \varphi_2^a \mid \ true \cup \varphi'^a$$

$$\varphi'^a ::= \neg ep \ \mid \varphi'^a \vee \varphi^a \ \mid \varphi_1'^a \wedge \varphi_2'^a \ \mid true \cup \varphi'^a$$

The language φ^a is LTL without $\cup$ and $\bigcirc$ operators for atomic propositions on states (sp). This includes invariant and safety properties on states. Atomic propositions on events (ep) occur only under a negation and only as an "eventually" formula (*i.e.,* in $true \cup \varphi'^a$). This language makes it possible to define liveness properties on $\neg ep$. For instance, the property $true \cup \neg print$ which is satisfied by the sequence

$$print : print : print : read : \epsilon : \ldots$$

is preserved by any aborter. An aborter will either leave the read instruction or abort the execution; in both cases, the current instruction will be eventually different from *print* (ϵ is not *print*). We assume here that ep cannot match ϵ; $true \cup \neg\epsilon$ would not preserved by an aborter stopping the program before the first instruction.

Many properties preserved by observer aspects are not preserved by aborters. Of course, this comes from their ability to abort programs. For example, $x = 0 \cup x = 1$ is satisfied by the following sequence

$$x = 0 : x = 0 : x = 1 : \epsilon : \epsilon : \ldots$$

but if an aborter aspect terminates the execution before $x = 1$ then the woven trace becomes

$$(x = 0, \texttt{abort}) : (x = 0, \epsilon) : (x = 0, \epsilon) : \ldots$$

and the property $x = 0 \cup x = 1$ is not satisfied anymore. On the other hand, properties of the form $x = 0 W x = 1$ are preserved.

The preservation of properties of Grammar 3.5 by aborter aspects is formalized by Theorem 3.6.

Theorem 3.6.

$$\begin{array}{l} \forall(C,\Sigma).\ \ \Sigma^{\psi} \in \mathcal{A}_a \ \ \Rightarrow \ \ \forall(p \in \varphi^a).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p \\ with \ \ \alpha = \mathcal{B}(C,\Sigma^b) \ \ and \ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma) \end{array}$$

Examples of aborters are security aspects that detect forbidden states or sequences of instructions or aspects that guarantee that a computation stops after a time-out. In general, an aspect which checks if a condition is violated by the base program and throw an exception without modifying the base state is an aborter. In [7], aspects are local security policies which can be woven on untrusted applets. Aspects only update their own state but abort the applet should it try to violate the policy. In [19], aspects are timed constraints which may terminate programs to guarantee availability of shared resources. In [2], Wampler presents a tool named Contract4J that takes invariants and generates aspects enforcing user-defined contracts. An aspect observes the execution and aborts it as soon as a contract is violated.

3.1.3 Confiners

An aspect is a confiner (Definition 3.7) if the state of any configuration of the woven program is a reachable state. In general, confiners can modify the control-flow and the state of the base program.

The set of reachable states from the configuration made of the program C and the state Σ^b is denoted by $Reach_b(C, \Sigma^b)$ with:

$$Reach_b(C, \Sigma^b) = \{\Sigma^{b'} \mid (C, \Sigma^b) \xrightarrow{*}_b (C', \Sigma^{b'})\}$$

Definition 3.7 formalizes the fact that the base state of any configuration in the woven trace is reachable by the base program.

Definition 3.7.

$$\forall (C, \Sigma).\ \Sigma^\psi \in \mathcal{A}_c \ \Leftrightarrow\ \forall (j \geq 1).\ \tilde{\alpha}_j = (i, \Sigma_j) \ \wedge\ \Sigma^b_j \ \in\ Reach_b(C, \Sigma^b)$$
$$\textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)$$

Observers and aborters are included in the category $\mathcal{A}_c$ of confiners. The set of properties preserved by confiners (Grammar 3.8) is a subset of the set of properties preserved by aborter aspects (Grammar 3.5).

Grammar 3.8.

$$\varphi^c ::= sp \ \mid \neg sp \ \mid \varphi^c_1 \vee \varphi^c_2 \ \mid \varphi^c_1 \wedge \varphi^c_2 \ \mid \varphi^c_1 W \mathit{false}$$

The language φ^c is restricted to invariant properties (*i.e.*, $\Box\varphi$ or $\varphi W \mathit{false}$) on states. Since confiner aspects can modify the control flow of events without restriction no properties involving atomic propositions on events in φ^c are preserved. For the same reason, safety properties such as $\varphi^c_1 W \varphi^c_2$ are not preserved by confiners. For example, the base program trace

$$x = 0 : x = 1 : x = 2 : \epsilon : \epsilon : \ldots$$

satisfies the safety property $x = 0 W x = 1$. However, after the weaving of a confiner that remains in $Reach_b$, the woven sequence can be

$$x = 0 : x = 2 : x = 0 : x = 1 : \epsilon : \ldots$$

which does not satisfies the safety property $x = 0 W x = 1$.

The preservation of properties of Grammar 3.8 by confiners is formalized by Theorem 3.9.

Theorem 3.9.

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_c \quad \Rightarrow \quad \forall(p \in \varphi^c).\ \alpha \models p \Rightarrow \tilde{\alpha} \models p$$
$$\textit{with}\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Examples of confiners are reset aspects that restore the initial state of the base program, fault-tolerance aspects that restore a safe execution state from a previous checkpoint, or memo aspects that shortcut a computation (or a already performed request) and returns its cached result. In all cases, in order to always remain in the reachable states, the reset (roll-back or caching) action must be considered as atomic. For example, a non-atomic roll-back is likely to create unreachable states in the middle of the restoration. A memo aspect is also likely to fail to change some temporary variables that are used when the result is not in the cache and must be computed. In such cases, aspects are confiners only if we restrict properties to a subset of the base program state. Without these restrictions, such aspects belong to the category presented next *i.e.,* weak intruders.

3.1.4 Weak intruders

An aspect is a weak intruder (definition 3.10) if states of a configuration with a current base program instruction (*i.e.,* i_b) are always reachable states. In other words, a weak intruder aspect may produce unreachable states during advice execution but always returns to reachable states when it returns to the base program. Confiners are special cases of the weak intruder aspect category.

Definition 3.10 formalizes the fact that the base state of any configuration with a current instruction i_b in the woven trace is reachable by the base program.

Definition 3.10.

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_w \ \Leftrightarrow\ \forall(j \geq 1).\ \tilde{\alpha}_j = (i_b, \Sigma_j) \ \Rightarrow\ \Sigma^b_j \ \in\ \mathit{Reach}_b(C,\Sigma^b)$$
$$\textit{with}\ \ \alpha = \mathcal{B}(C,\Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Since a weak intruder can modify the control-flow and the state of the base program, it can violate invariants during the execution of advice. There is no LTL property preserved for all weak intruders and programs. However, if the (weaving of) weak intruder aspect terminates (definition 3.11) then it preserves properties of the form $\Diamond\varphi^c$. That is, the woven program eventually preserves invariant properties (*i.e.,* after the last advice).

Definition 3.11.

$$\forall(C,\Sigma).\ \Sigma^{\psi}\ \textit{terminates} \ \Leftrightarrow\ \exists(j \geq 1).\forall(k > j).\ \tilde{\alpha}_k = (i_b, \Sigma_k)$$
$$\textit{with}\ \ \alpha = \mathcal{B}(C,\Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

For example, the base program trace

$$x = 0 : x = 1 : x = 0 : (\epsilon, x = 1) : (\epsilon, x = 1) : \ldots$$

satisfies the φ^c property $(x = 0 \vee x = 1)W\mathit{false}$. The woven sequence

$$x = 0 : x = 1 : x = 0 : x = 2 : (\epsilon, x = 0) : (\epsilon, x = 0) : \ldots$$

violates the property when $x = 2$ (a possible state produced during the execution of an advice). However, the final configuration $(\epsilon, x = 0)$ has a state $(x = 0)$ reachable by the base program. So, $(x = 0 \vee x = 1)W false$ is eventually satisfied (*i.e.,* $\Diamond((x = 0 \vee x = 1)W false)$).

Theorem 3.12 formalizes the fact that if the base program satisfies an invariant property p then the woven execution with a terminating weak intruder aspect satisfies eventually p.

Theorem 3.12.

$$\forall(C,\Sigma).\Sigma^{\psi} \in \mathcal{A}_w \ \wedge\ \Sigma^{\psi} \textit{ terminates } \Rightarrow\ \forall(p \in \varphi^c).\ \alpha \models p \ \Rightarrow\ \tilde{\alpha} \models \Diamond p$$
$$\textit{with } \ \alpha = \mathcal{B}(C,\Sigma) \ \textit{ and } \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)$$

Fault tolerant aspects performing non atomic rollbacks are typical weak intruder aspects. They may produce unreachable states during advice execution (*i.e.,* the rollback) but eventually reach a previous safe state. Similarly, aspects performing non atomic resets are weak intruders.

3.2 Non-Deterministic Case

Non-determinism brings two new aspect categories: *selectors* ($\mathcal{A}^*_s$) which select some executions among the set of possible executions, and *regulators* ($\mathcal{A}^*_r$) which can select but also abort executions.

The categories of observers, aborters, selectors, regulators, confiners and weak intruders form a hierarchy

$$\mathcal{A}^*_o \subset \mathcal{A}^*_a \subset \mathcal{A}^*_r \subset \mathcal{A}^*_c \subset \mathcal{A}^*_w \qquad \mathcal{A}^*_o \subset \mathcal{A}^*_s \subset \mathcal{A}^*_r$$

where aborters $\mathcal{A}^*_a$ and selectors $\mathcal{A}^*_s$ cannot be compared. Properties are defined using CTL* which permits to quantify formulae over the set of execution traces. This logic is strictly more expressive than LTL. The classes of properties $\theta^o, \theta^a, \theta^s, \theta^r, \theta^c, \theta^w$ preserved by the corresponding aspect categories are related by a dual inclusion hierarchy.

As in the deterministic case, an observer does not modify the control-flow and the state of the base program. In particular, the woven and the base program have the same set of traces of base instructions (*i.e.,* after projection by $proj_b$). The examples of aspects discussed before remain valid in the non-deterministic case. For instance, security aspects are also aborter aspects for non-deterministic programs. Each class of preserved properties in the non-deterministic case generalizes its deterministic version (*e.g.,* θ^o is strictly more expressive than φ^o). In this section, we do not (re)present all categories but focus instead on the two new categories (selectors and regulators) and their corresponding classes of properties.

3.2.1 Selectors

A selector does not modify the state of the base program. However, a selector can modify the control-flow of the base program by selecting a subset of execution traces among the set of all possible execution traces. Obviously, this

new category of aspect only makes sense for non-deterministic programs since its effect is to suppress some non-deterministic choices.

A selector (Definition 3.13) cannot introduce new execution traces: for any trace in the set of woven executions, there exists a trace in the set of base executions with the same sequence of base instructions (*i.e.,* related by $proj_b$).

Definition 3.13.

$$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_s^* \ \Leftrightarrow\ \forall(\tilde{\alpha} \in \mathcal{W}^*(C,\Sigma)).\exists(\alpha \in \mathcal{B}^*(C,\Sigma^b)).\ proj_b(\tilde{\alpha}) = proj_b(\alpha)\ \wedge\ preserve_b(\tilde{\alpha})$$

The properties defined by θ^s in Grammar 3.14 are preserved by selectors.

Grammar 3.14.

$$\theta^s ::= sp \mid \neg sp \mid \theta_1^s \vee \theta_2^s \mid \theta_1^s \wedge \theta_2^s \mid \forall\omega^s$$

$$\omega^s ::= \theta^s \mid \omega_1^s \vee \omega_2^s \mid \omega_1^s \wedge \omega_2^s \mid \omega_1^s \cup \omega_2^s \mid \omega_1^s W \omega_2^s \mid true \cup \omega'^s$$

$$\omega'^s ::= ep \mid \neg ep \mid \theta^s \mid \omega_1'^s \vee \omega_2'^s \mid \omega_1'^s \wedge \omega_2'^s \mid \omega_1^s \cup \omega_2^s \mid \omega_1^s W \omega_2^s \mid true \cup \omega'^s$$

Grammar 3.14 can be described as a generalization to CTL* of the class preserved by observers (*i.e.,* φ^o). It does not include the $\exists$ operator because an execution of the base program that satisfies a property $\exists\omega$ can be removed by a selector. The preservation of θ^s by selectors is expressed by the theorem 3.15.

Theorem 3.15.

$\forall(C,\Sigma).\ \Sigma^{\psi} \in \mathcal{A}_s^* \ \Rightarrow\ \forall(p \in \theta^s).\forall(\alpha \in \Gamma).\Gamma, \alpha_1 \models p \Rightarrow \forall(\tilde{\alpha} \in \tilde{\Gamma}).\tilde{\Gamma}, \tilde{\alpha}_1 \models p$
where $\Gamma = \mathcal{B}^*(C,\Sigma^b)$ *and* $\tilde{\Gamma} = \mathcal{W}^*(C,\Sigma)$

Examples of selectors are scheduling aspects or refinement aspects that removes some non-determinism. The scheduling aspects of [18] specify and enforce scheduling policies to networks of communicating processes. A scheduling aspect selects a subset of desired execution traces out of the set of all possible interleavings. These aspects are typical selectors.

3.2.2 Regulators

Regulators are both aborters and selectors. A regulator (Definition 3.16) does not modify the state of the base program ($preserve_b$). However, it can modify the control-flow of the base program, either as an aborter by aborting the program or, as a selector by selecting a subset of the execution traces. For any trace $\tilde{\alpha}$ of the woven program executions:

- either there exists a trace α among the base executions that has the same base instructions as $\tilde{\alpha}$ (*i.e.,* the aspect does not modify the control-flow of the base program);
- or there exists a prefix $\alpha_{\rightarrow i}$ in a base execution trace and a prefix $\tilde{\alpha}_{\rightarrow j}$ in the woven execution trace that have the same base instructions and the rest of the woven trace has only final instructions ϵ.

Definition 3.16.

$$\begin{array}{rcl} \forall(C,\Sigma).\Sigma^{\psi} \in \mathcal{A}_r^* & \Leftrightarrow & \forall(\tilde{\alpha} \in \mathcal{W}^*(C,\Sigma)).\exists(\alpha \in \mathcal{B}^*(C,\Sigma^b)). \\ & & \quad preserve_b(\tilde{\alpha}) \ \wedge \ proj_b(\tilde{\alpha}) = proj_b(\alpha) \\ & \vee & \exists(i \geq 0).\ \exists(j \geq i).proj_b(\alpha_{\rightarrow i}) = proj_b(\tilde{\alpha}_{\rightarrow j}) \\ & & \quad \wedge \ \forall(k > j).\ \tilde{\alpha}_k = (\epsilon, \_) \end{array}$$

Note that, this definition does not relate all base execution traces with a woven one, since regulator aspect can select out base execution similarly to selector aspects.

The properties defined by θ^r in Grammar 3.17 are preserved by regulator aspects.

Grammar 3.17.

$$\begin{array}{rcl} \theta^r & ::= & sp \mid \neg sp \mid \theta_1^r \vee \theta_2^r \mid \theta_1^r \wedge \theta_2^r \mid \forall\omega^r \\ \omega^r & ::= & \theta^r \mid \omega_1^r \vee \omega_2^r \mid \omega_1^r \wedge \omega_2^r \mid \omega_1^r W \omega_2^r \mid true \cup \omega'^r \\ \omega'^r & ::= & \neg ep \mid \omega'^r \vee \theta^r \mid \omega_1'^r \wedge \omega_2'^r \mid true \cup \omega'^r \mid \forall\omega'^r \end{array}$$

Grammar 3.17 can be seen as the intersection of the class of properties preserved by selectors (*i.e.,* θ^s) and the class preserved by aborters (*i.e.,* θ^a, the generalization of φ^a).

As before, the $\exists$ operator is excluded since a regulator aspect may remove execution traces from the set of all possible traces. The state properties of the form $\omega_1^r \cup \omega_2^r$ are not preserved since the aspect may abort the program before ω_2^r. As far as event properties are concerned, only liveness properties involving $\neg ep$ are preserved. For example, $true \cup \neg ep$ is preserved since if the aspect aborts the execution $\neg ep$ will be satisfied after abortion (*i.e.,* when the configuration becomes $(\epsilon : \bullet, \Sigma)$).

The preservation of θ^r by regulative aspects is expressed by Theorem 3.18.

Theorem 3.18.

$\forall(C,\Sigma).\Sigma^{\psi} \in \mathcal{A}_r^* \ \Rightarrow \ \forall(p \in \theta^r).\forall(\alpha \in \Gamma).\Gamma, \alpha_1 \models p \Rightarrow \forall(\tilde{\alpha} \in \tilde{\Gamma}).\tilde{\Gamma}, \tilde{\alpha}_1 \models p$
where $\Gamma = \mathcal{B}^*(C,\Sigma^b)$ *and* $\tilde{\Gamma} = \mathcal{W}^*(C,\Sigma)$

3.3 Interactions between Aspects

We study in this section the issues raised by the composition of several aspects. For simplicity reasons, the aspect function Σ^{ψ} (see Sec. 2.1) does not distinguish between a single or several aspects; it just inserts an advice at the appropriate place.

In the following, we write $A_1; A_2$ for the composed aspect where A_1 has precedence when both match the same join point. For example, the composition of two before aspects $\Sigma^{\psi} = A_1; A_2$ is such that when A_1 and A_2 match the same instruction i then $\Sigma^{\psi}(i : C, \Sigma) = (a_1 : a_2 : i : C, \Sigma)$ with a_1 (resp. a_2) denoting the advice of A_1 (resp. A_2). The description of the composition of around aspects requires a proceed stack to store the code to be executed when a proceed instruction is called. For a formal treatment of aspect composition see [13].

Even if the framework of Section 2 is too abstract to represent explicitely aspect composition, we discuss informally two issues:

- the composition of two aspects. In particular, knowing the categories of two aspects, can we determine the category of their composition?

- the commutativity of weaving. In particular, are there categories of aspects ensuring that the weaving of two aspects can be performed in any order ?

3.3.1 Composition

At first sight, the composition of two aspects $A_1 \in \mathcal{A}_x$ and $A_2 \in \mathcal{A}_y$ with $\mathcal{A}_x \subseteq \mathcal{A}_y$ should belong to $\mathcal{A}_y$. That is to say, the category of $A_1; A_2$ should be the largest (less constrained) category of the two aspects. For instance, the composition of two observers should be an observer, or the composition of an observer and an aborter should be an aborter. However, some precautions should be taken.

First, we must assume that an aspect cannot modify the local state of another aspect. Observers and aborters, whose advice must always returns to the base program, require another constraint. Indeed, the composition of two observers A_1 and $A2$ can produce a non-terminating aspect.

Consider, for example, the aspect

$$A_1 : \texttt{before foo}(*)\ \ \mathtt{n}_{\mathtt{A1}} := \texttt{bar}(\mathtt{n}_{\mathtt{A}_1})$$

matching calls to `foo` and updating its local state using the function `bar` and the aspect A_2

$$A_2 : \texttt{before bar}(*)\ \ \mathtt{n}_{\mathtt{A2}} := \texttt{foo}(\mathtt{n}_{\mathtt{A}_2})$$

matching calls to `bar` and updating its local state using the function `foo`. Assuming that `foo` and `bar` are pure terminating functions, both aspects are observers. But the weaving of $A_1; A_2$ loops as soon as a call to `foo` or `bar` is encountered; the execution never returns to the base program. One should ensure that no cycle can occur in the composition of aspects. This can be done by analysis the aspects' pointcuts and advice. A sufficient restriction is to enforce that aspects can only match base instructions.

These constraints ensure that different observers/aborters are independent. Weaving two observers (resp. an observer and an aborter or two aborters) $A_1; A2$ can be seen as weaving a single observer (resp. aborter). Even if our framework is too abstract to treat this issue rigorously, we believe that, a composition of aspects should belong to the most expressive category involved.

3.3.2 Commutativity

If two aspects never match the same join point then their weaving order is irrelevant. In [15, 16], we propose an analysis to determine whether two aspects are independent *i.e.,* never match the same join point.

When two aspects match the same join point, the weaver usually relies on a precedence relation to ensure a deterministic behavior. The question here is whether such precedence is still necessary with our restricted categories of aspects

The answer depends on the definition of commutativity or equivalence between programs. If we consider trace equivalence, then as soon as two aspects match the same join point, their weaving never commutes. Executing A_1 before A_2 produces a different trace than the other way around.

A more relaxed definition of program equivalence is to enforce that the traces after projection by $proj_b$ are identical and the states of the base program and aspects are identical at each base instruction. This ensures that the base program and the aspects computes the same results. Even with this relaxed notion, the weaving of two observers does not commute. Consider for instance the following two observers

$$A_1 : \texttt{before foo}(*) \ \ \texttt{n} := \texttt{n} + \texttt{1}$$

matching calls to `foo` and incrementing its local variable `n` and the aspect A_2

$$A_2 : \texttt{before foo}(*) \ \ \texttt{b} := (\texttt{n} > \texttt{0})$$

matching calls to `foo` and setting its local variable `b` to true if $n > 0$. Then, assuming an intial state of A_1 where $n = 0$, the first call to `foo` will change the state of A_2 to $b = \mathit{true}$ or $b = \mathit{false}$ depending on the precedence. This comes from the fact that an aspect can read the local state of another one.

Consider now the observer

$$A_1 : \texttt{before } (\texttt{foo}(\beta) \wedge \beta \neq \texttt{0}) \ \ \texttt{foo(0)}$$

matching calls to `foo` with a non null parameter and calling the function `foo(0)` and the observer

$$A_2 : \texttt{before } \ foo(\beta) \ \ n := \beta$$

matching all calls to `foo` and updating its local variable `m` to the value of the parameter. The sequence of advice executed before the call `foo(1)` will be either $\texttt{n} := \texttt{0}; \texttt{foo(0)}; \texttt{n} := \texttt{1}$ or $\texttt{n} := \texttt{0}; \texttt{n} := \texttt{1}; \texttt{foo(0)}$ depending on precedence. The local state of A_2 varies depending on the weaving order. This comes from the fact that an aspect can match the advice of another one.

Two conditions ensure that the weaving of two observers commutes:

1. the observers cannot read the local state of each other;

2. the observers cannot match an instruction of each other.

With these restrictions, observers are *semantically independent*: their execution are unaffected by the weaving of another observer and therefore weaving commutes. But still, the other categories do not commute. For example, weaving an aborter before an observer may prevent the observer to execute its advice compared to the other weaving order. The observer's final local state will differ depending on which is woven first.

Another even more relaxed definition of equivalence is to enforce that traces after projection on base instructions and base states are identical. This ensures that the effect of aspects on the base program are equivalent regardless of the weaving order. With this definition, two observers commute since, even if they may influence each other, they cannot change the base program's control flow and state. In general, the weaving of an observer and aborter does not commute. For example, an aborter may terminate the program depending on the

observer's local state. However, with the restrictions (1) and (2) above, an observer commutes with any other aspect, an aborter commutes with any other aspect which does not change the base state. Selectors do not commute since they are in competition to select a non deterministic choice and therefore precedence matters. Confiners (or weakly invasive aspects) do not commute since they share (read and write) access to the base state.

4 Specialized Aspects Languages

In this section, we present specialized aspects languages for our different classes of properties. We choose a simple, expressive enough and standard imperative language as our base language (Section 4.1). It is very close to languages used in formal semantics books such as the *IMP* language in [40] or the *While* language in [34]). We present a generic pointcut language shared by our aspect languages in (Section 4.2). These languages differ by the more or less restrictive constraints on their advice. We introduce in Sections 4.3.1, 4.3.2 and 4.3.3 the constraints corresponding to the observer, aborter and confiner categories. All aspects defined in a language belong to the corresponding category. Therefore each language ensures the preservation of the corresponding class of properties by construction. In Section 4.4, we extend our base language with a non deterministic statement and we present two aspect languages (selector and confiner) specialized for non determinism.

4.1 Base Language

A base program P is a sequence D of declarations of global variables (`var` g) and procedures (`proc` I) followed by a main statement S. Statements comprise usual commands (assignment, procedure call, sequencing, conditional, while loop), the instruction `abort` that ends a program execution, `skip` that does nothing and `loop`(A) S that repeats A times the statement S. Arithmetic and boolean expressions are described by nonterminals A and B respectively. There are two distinguished kinds of variables:

- global variables (g) which are declared in D;
- local variables (l) declared as parameters of procedures.

Both kinds of variables can be used in assignments and expressions.

Grammar 4.1.

$$
\begin{array}{lcl}
P & ::= & D\ \{S\} \\
D & ::= & \texttt{var}\ g{:=}A \mid \texttt{proc}\ I(l_1,\ldots l_n)\ S \mid D_1;D_2 \\
S & ::= & V{:=}A \mid I(A_1,\ldots A_n) \mid S_1;S_2 \mid \texttt{if}(B)\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}(B)\ S \\
 & & \texttt{abort} \mid \texttt{skip} \mid \texttt{loop}(A)\ S \\
A & ::= & n \mid V \mid A_1 + A_2 \\
B & ::= & \mathit{true} \mid A_1{=}A_2 \mid A_1{<}A_2 \mid B_1\&B_2 \mid\ !B \\
V & ::= & g \mid l \\
I & ::= & p
\end{array}
$$

We consider only integer variables to avoid typing issues. However, the language could be easily extended and equipped with a type system. The operational semantics of this language is very similar to the *While* language of [34]. As required by our framework (Section 2.1), its semantics is defined by a relation $\rightarrow_b$ on configurations (C, Σ^b) where C is a sequence of statements and Σ^b is made of an environment associating global variables and parameters to their values and of a return stack used by procedure calls and returns. It is described in details in appendix B.

Example 4.2 illustrates the base language with a simple program which will be used throughout.

Example 4.2. *The following program computes the fourth fibonacci number in the variable* `result`*:*

```
var result := 0;
proc fib(x)
    if(x = 0) then result := result + 1 else
    if(x = 1) then result := result + 1
    else  fib(x − 1); fib(x − 2)
{fib(4)}
```

4.2 Generic Pointcut Language

Our aspect languages share the same pointcut language which is defined by grammar 4.3.

Grammar 4.3.

$$
\begin{array}{lcl}
P & ::= & S^p \mid if(B^p) \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \\
S^p & ::= & V^p{:=}A^p \mid I^p(A^p_1, \ldots, A^p_n) \mid S^p_1; S^p_2 \mid \\
 & & \texttt{if}(B^p)\ \texttt{then}\ S^p_1\ \texttt{else}\ S^p_2 \mid \texttt{while}(B^p)\ S^p \mid \\
 & & \texttt{abort} \mid \texttt{skip} \mid \texttt{loop}(A^p)\ S^p \mid \beta_S \mid \neg S^p \\
A^p & ::= & n \mid V^p \mid A^p_1 + A^p_2 \mid \beta_A \mid \neg A^p \\
B^p & ::= & true \mid A^p_1{=}A^p_2 \mid A^p_1{<}A^p_2 \mid B^p_1 \& B^p_2 \mid !B \mid \\
 & & \beta_B \mid \neg B^p \\
V^p & ::= & g \mid l \mid \beta_V \mid \neg V^p \\
I^p & ::= & p \mid \beta_I \mid \neg I^p
\end{array}
$$

A pointcut is either a statement with pattern variables S^p (a static pointcut), or a predicate $if(B^p)$ (a dynamic pointcut), or a logical composition of pointcuts. A statement pattern S^p is a statement which enables, for each syntactic category (expressions, variables, ...), pattern variables as well as negative patterns (*e.g.,* $\neg S$). For example, A^p defines patterns on arithmetic expressions with pattern variables (β_A) (able to match any arithmetic expression) and negations. I^p defines patterns of procedure identifiers. Matching of a pattern S^p *w.r.t.* a current instruction assigns values to pattern variables β_S, β_A, ... These values will be substituted for the occurrences of pattern variables occurring in dynamic pointcuts $if(b)$ as well as in advice. The semantics of patterns with negation (called anti-patterns) is described in details in [27].

Dynamic pointcuts $if(b)$ should represent valid boolean expressions after substitution. To ensure this property, negation of patterns (*e.g.,* $\neg B^p$) are not allowed to occur within dynamic pointcuts. Also, variables occurring in dynamic pointcuts (and advice) should also occur outside the scope of a negation in the static pointcut (to have a unique substitution).

Example 4.4. *To provide some intuition, here are a few examples of patterns:*

- $\texttt{x} := \beta_A$ *matches all assignments to* `x`*;*
- $(\neg\texttt{x}) := \beta_A$ *matches all assignments but those to* x*;*
- $\neg(\texttt{x} := \texttt{y})$ *matches all statements but* $\texttt{x} := \texttt{y}$*;*
- $\texttt{while}(\beta_B)\ \beta_S$ *matches all while statements;*
- $\texttt{p}(\texttt{3},\beta_A) \wedge \texttt{if}(\beta_A = \texttt{0})$ *matches all calls to* `p` *with* `3` *and an arithmetic expression whose value is* `0`*.*

Our implementation of pointcuts relies on a preliminary transformation described in [13]. A pointcut p is transformed into an equivalent pointcut of the form

$$(p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n))$$

where the static patterns p_i are *mutually exclusive*. Each static pattern is matched to the current instruction using the anti-pattern algorithm [27] written $match^s$ until a match is found. The function $match^s$ returns a substitution which is applied to the corresponding dynamic pointcut and advice that will be evaluated relatively to the state. If no match exists, the function $match^s$ returns *Fail*. For instance, $match^s(\texttt{p}(\texttt{3},\beta_A),\texttt{p}(\texttt{3},\texttt{0}))$ returns $[\beta_A \mapsto 0]$ and $match^s(\neg\beta_A,\texttt{0})$ returns *Fail*.

4.3 Aspects for deterministic languages

In this section, we define three restricted aspect languages that ensures that aspects defined in these languages are observers, aborters and confiners respectively. The two first languages are general purpose; they can be used to describe any king of observers or aborters. The last one is a confiner language dedicated to memoization.

4.3.1 Observer language

As seen in Section 3.1.1, an observer does not modify the control flow of the base program but only inserts advice instructions (i_a). In order to remain consistent with AspectJ and most aspect-oriented languages, we consider around aspects composed of an arbitrarily complex statement of i_a instructions, followed by the command `proceed` to execute the matched statement, followed by another arbitrarily complex statement of i_a. When the advice execution is over, the base program execution is resumed after the matched statement.

Note that our `proceed` instruction does not have parameters. Otherwise, observers would be able to modify the parameters of procedures and arbitrarily change the state or the control-flow of the base program. Furthermore, the advice should terminate, otherwise the base program execution is never resumed and its control flow is not preserved. We ensure termination by disallowing while statements in advice, checking that there is no loop in the call graph of advice and ensuring that the pointcut cannot match any statement of its own advice.

Another option would be to permit while-loops and recursion in advice and make the programmer responsible for ensuring termination.

The second condition an observer should obey is not to modify the state of the base program (*i.e.,* i_a instructions should not change the state Σ^b). We distinguish the base program variables (that can be read by an advice) from the aspect variables (that can be read *and* written by a i_a instruction).

The semantics of `proceed` is expressed using a proceed stack (written Σ^P) in the global state(see [13]). When an around advice applies, the matched instruction is pushed onto that stack. The `proceed` instruction pops and executes the instruction on top on the proceed stack:

$$\textsc{Proceed}\quad \frac{\Sigma^P = i : \Sigma'^P}{(\texttt{proceed} : C, X \cup \Sigma^P) \rightarrow (i : C, X \cup \Sigma'^P)}$$

The syntax of observers is defined by the Grammar 4.5.

Grammar 4.5.

$$\begin{array}{lcl}
Asp^o & ::= & D^o \texttt{ around } P\ \{S_1^o; \texttt{proceed};\ S_2^o\} \\
D^o & ::= & \texttt{var } g^o := A^o \mid \texttt{proc } I^o(l_1^o, \ldots, l_n^o)\ S^o \mid D_1^o; D_2^o \\
S^o & ::= & V^o{:=}A^o \mid I^o(A_1^o, \ldots, A_n^o) \mid S_1^o; S_2^o \mid \texttt{skip} \mid \\
 & & \texttt{if}(B^o)\ \texttt{then}\ S_1^o\ \texttt{else}\ S_2^o \mid \texttt{loop}(A^o)\ S^o \\
A^o & ::= & n \mid V' \mid A_1^o + A_2^o \mid \beta_{\text{A}} \\
B^o & ::= & \mathit{true} \mid A_1^o{=}A_2^o \mid A_1^o{<}A_2^o \mid B_1^o \& B_2^o \mid\ !B^o \mid \beta_{\text{B}} \\
V^o & ::= & g^o \mid l^o \\
V' & ::= & V^o \mid g \mid \beta_{\text{V}} \\
I^o & ::= & p^o
\end{array}$$

An observer Asp^o defines variables g^o and procedures p^o to form the local state of the aspect. Then, `around` associates a pointcut with an advice which contains exactly one `proceed`. We have considered that an aspect has one pointcut and one advice to simplify the presentation but this could be easily generalized to several pointcuts and advices. The declarations D^o must not contain any occurrence of pattern variables. Other statements S^o are similar to statement patterns S^p but without negation $\neg$. Indeed, an advice must be a valid executable code after substitution of its pattern variables (β_{A}, β_{B}, β_{V}). Note that, the statement `abort` is not allowed in advice since it would change the control flow of the base program. Similarly, pattern variables β_{S} for statements are forbidden since they could be used to execute assignments to base program variables in the advice. Note that, assignment statements in advice can only modify variables of the aspect (V^o). Of course, aspect and base variables (V') can both be read. Finally, an advice can only call procedure defined in the aspect (I^o) since calling a base program procedure could modify the base program state.

An aspect that counts calls to `fib` (Example 4.2) is defined in Example 4.6. This profiling aspect respects the grammar Asp^o and is therefore an observer.

Example 4.6. *Profiling calls to* `fib`

$$\texttt{var n} := \texttt{0 around } (\texttt{fib}(\beta_{\text{A}}))\ \texttt{n} := \texttt{n} + \texttt{1}$$

The semantics of weaving (Section 2.1) represents an aspect as a function Σ^{ψ} that takes the current configuration (C, Σ) as parameter and returns either a new woven configuration (C', Σ'), or *nil* when the pointcut does not match. We define the semantics of our aspect language in order to generate Σ^{ψ} from an aspect definition as follows. The resulting function takes the current configuration as parameter and matches the first instruction i. First, as mentioned in the previous section, the pointcut p of the aspect is transformed into an exclusive disjunction of the form $(p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n))$. The function tests if the current instruction i is matched one of the static pointcuts p_i. If i is not matched, the function returns *nil*. Otherwise, the current instruction i is replaced by a code a and i is pushed on the proceed stack Σ^P. When it is executed, the conditional a tests the dynamic part b_i of the matched pointcut. If b_i is satisfied the advice s is executed, otherwise the execution `proceed`s with the original instruction i (the advice is not executed). The pattern variables in b and s are substituted by their matched values using the substitution σ returned by match^s.

$$
\begin{array}{l}
[\![\texttt{around}\ (p)\ s]\!] = \\
\mathit{let}\ (p_1 \wedge \mathit{if}(b_1)) \vee \ldots \vee (p_n \wedge \mathit{if}(b_n)) = \mathit{Transf}(p)\ \mathit{in} \\
\quad \lambda(i : C, X \cup \Sigma^P).\quad \mathit{case}\quad \begin{array}{lcl}
\mathit{match}^s(p_1, i) = \sigma_1 & \mapsto & (\bar{a_1} : C, X \cup \bar{i} : \Sigma^P) \\
\ldots & & \\
\mathit{match}^s(p_n, i) = \sigma_n & \mapsto & (\bar{a_n} : C, X \cup \bar{i} : \Sigma^P) \\
\mathit{otherwise} & \mapsto & \mathit{nil}
\end{array} \\
\quad \mathit{where}\ a_i = \sigma_i(\texttt{if}(b_i)\ \texttt{then}\ s\ \texttt{else proceed})
\end{array}
$$

The instruction $\bar{i}$ and the conditional $\bar{a_i}$ are tagged (see Section 2.1) to prevent infinite weaving by matching them again and again.

The semantics distinguishes evaluation of the static part of a pointcut from the evaluation of its dynamic part. This faithfully models AspectJ-like languages where the dynamic part of a pointcut can depend on a previous advice execution. Property 4.7 formalizes the fact that any aspect in Asp^o is an observer.

Property 4.7. $\forall a \in \mathit{Asp}^o.[\![a]\!] \in \mathcal{A}_o$

A sketch of the proof can be found in the appendix.

4.3.2 Aborter language

An aborter is an observer which may abort the execution. The aborter language is therefore very similar to the observer language. Its grammar Asp^a is expressed exactly as Asp^o except that the statement `abort` is allowed in S^a. The `abort` instruction reduces any configuration in a final configuration (see Section 3.1.2).

Example 4.8 specifies an aspect counting the number of calls to the procedure `fib` (of the Example 4.2). If the number of calls reaches 100.000 the program is aborted. This aspect can be used to enforce some computation quota. It is defined in Asp^a so it is an aborter.

Example 4.8. *Regulating calls to* `fib`

```
var nbCalls := 0; around (fib(β_A))
nbCalls := nbCalls + 1;
if(nbCalls = 100000) then abort else skip;
proceed; skip
```

Property 4.9 states that any aspect in Asp^a is an aborter.

Property 4.9. $\forall a \in Asp^a.[\![a]\!] \in \mathcal{A}_a$

The proof is very close to the proof of Property 4.7.

4.3.3 A confiner language

Confiners can arbitrarily modify the control flow and the state of the base program as long as the base state remains in the set of originally reachable states. A general purpose language ensuring this property is very hard to design. However, two specialized confiner languages come to mind:

- optimization dedicated languages whose advice would jump directly to a future reachable state;
- fault-tolerance dedicated languages whose advice would roll-back to a previous reachable state.

Fault tolerance makes sense for a deterministic program when the runtime environment is non deterministic (*i.e.,* faults can occur). In the next section, we discuss about an aspect language for fault tolerance for non deterministic programs. We propose here a specialized language dedicated to defining *memo aspects*. A memo aspect is an optimizing aspect that caches computations. It introduces memoization in the woven program: when a computation is performed for the first time, it stores its arguments and results. When the same computation is performed again, it shortcuts it and directly returns its previously stored results. Grammar 4.10 presents the syntax of this language.

Grammar 4.10.

$$\begin{array}{ll} Asp^m ::= & \mathsf{memo}\ (I^m(A_1^p, \ldots, A_n^p) \wedge if(B^o)) \\ I^m ::= & p \mid \beta_I \end{array}$$

A memo aspect is a primitive memo applied to a pointcut whose static part denotes the procedure calls to be memoized, and dynamic part is an arbitrary predicate. In order to implement sophisticated strategies of memoization a memo aspect can be combined with an observer. Since observers (and aborters) are included in the confiner category, the composition of a confiner aspect with any observer (aborter, confiner) aspect is also a confiner. For example, the base program could be first woven with an observer that collects statistics regarding procedure calls (*e.g.,* number of calls, depth of recursion,...) in its variables. It is then woven with a memoization aspect whose predicate accesses the variables holding statistics.

To give the semantics of a memoization aspect, we need to compute the lists of variables a procedure reads and writes. These two lists are computed by the functions *read* and *write*. We can now define the semantics of a memo aspect as a program transformation taking the aspect and the declarations (D) of the

base program:

$$
\begin{array}{l}
T[\![\texttt{memo}\ (p(a_1,\ldots,a_n) \wedge \mathit{if}(B^o))]\!]D = \\
\texttt{var cache} := \texttt{empty} \\
\texttt{around}\ (p(a_1,\ldots,a_n) \wedge \mathit{if}(B^o)) \\
\texttt{if contain}(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p) \\
\texttt{then}\quad \mathit{write}[\![D]\!]p := \texttt{lookup}(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p) \\
\texttt{else}\quad \texttt{proceed}; \\
\qquad\quad \texttt{store}(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p, \mathit{write}[\![D]\!]p)
\end{array}
$$

A memo aspect defines an initially empty `cache` variable to store computation results. A cache entry associates a triplet $(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$ (a procedure identifier, the list of its arguments and the list of the variables read) to the list of values of its written variable $\mathit{write}[\![D]\!]p$.

When the pointcut is matched, the resulting substitution σ is applied to the advice and it fully instantiates the procedure, its arguments, as well as the lists of read ($\mathit{read}[\![D]\!]p$) and written ($\mathit{write}[\![D]\!]p$) variables. When the advice is executed, if the cache contains the result of the computation ($\texttt{contain}(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$) then the written variables are assigned with the result stored in the cache ($\texttt{lookup}(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p)$), else the computation is performed and the cache is updated ($\texttt{store}(p, a_1 : \ldots : a_n, \mathit{read}[\![D]\!]p, \mathit{write}[\![D]\!]p)$). Actually, such an aspect is a confiner only if the updating ($\mathit{write}[\![D]\!]p := \texttt{lookup}(...)$) is considered as atomic. Otherwise the updating of several variables produces temporary unreachable states. In a concurrent context, updating should also be atomic.

Note that, for the sake of conciseness, we have defined the advice in the base language extended with data structures (*i.e.,* `cache` implements a hash table, and lists to represent the values of read and written variables) and a return value for procedures (*e.g.,* `contain`, `lookup`).

Example 4.11 defines a memo aspect for the `fib` procedure defined in the Example 4.2. It is easy to check that the procedure `fib` reads no variable and writes the single variable `result`.

Example 4.11. *Memoizing* `fib`

$$\texttt{memo}\ (\texttt{fib}(\beta_A) \wedge \mathit{if}(\beta_A > 10))$$

This aspect memoizes calls to `fib` *only if its argument is greater than 10 (to amortize the cost of caching). The program transformation T would generate the following lower level aspect:*

$$
\begin{array}{l}
\texttt{var cache} := \texttt{empty} \\
\texttt{around}\ (\texttt{fib}(\beta_A) \wedge \mathit{if}(\beta_A > 10)) \\
\texttt{if}(\texttt{contain}(\texttt{fib}, [\beta_A], [])) \\
\texttt{then result} := \texttt{lookup}(\texttt{fib}, [\beta_A], []) \\
\texttt{else proceed}; \texttt{store}(\texttt{fib}, [\beta_A], [], [\texttt{result}])
\end{array}
$$

Our version of `fib` *(Example 4.2) computes many times the same calls and has exponential time complexity. The previous memo aspect suffices to change its complexity to linear time.*

4.4 Aspects for non deterministic languages

Non-deterministic (or concurrent) programs bring new interesting categories of aspects and classes of properties. In particular, Section 3.2 presents the categories of selectors and regulators.

Here we extend our base language with a non deterministic construct and we present two specialized aspect languages taking into account this extension.

4.4.1 Extension of the base language

We extend the imperative base language of the Section 4.1 with the non deterministic statement S_1 `or` S_2. This new statement executes non-deterministically either S_1 or S_2. Its semantics is defined by the two transition rules:

$$\text{OR}_1 \quad \frac{}{(S_1 \texttt{ or } S_2 : C, \Sigma) \rightarrow_b (S_1 : C, \Sigma)}$$

and

$$\text{OR}_2 \quad \frac{}{(S_1 \texttt{ or } S_2 : C, \Sigma) \rightarrow_b (S_2 : C, \Sigma)}$$

The syntax of pointcut statements (Grammar 4.3) is extended with the corresponding pattern S_1^P `or` S_2^P.

Observers and aborters as decribed in Section 4.3 apply to the new base language without any further extension than adding S_1^p `or` S_2^p to the pointcut language. Regarding our memo aspects, the functions *read* and *write* must be extended in order to collect variables in both branches of non-deterministic `or` statements. As in the deterministic case, this static analysis of read and written variables always terminates.

We now present two aspect languages dedicated with feature specific to non-determinsm: a selector and a weak intruder language.

4.4.2 A selector language

Selectors are observers that can select some executions in the set of all possible executions. In order to define the language Asp^s, we extend the advice language of observers by replacing the instruction `proceed` by the following non-terminal:

$$P^s ::= \texttt{proceedLeft} \mid \texttt{proceedRight} \mid \texttt{proceed} \mid \texttt{if}(B^o) \texttt{ then } P_1^s \texttt{ else } P_2^s$$

When the non deterministic instruction S_1 `or` S_2 is at the top of the proceed stack Σ^P, the instruction `proceedLeft` executes the left hand side S_1 (Rule ProceedLeft), and `proceedRight` executes the right hand side S_2 (Rule ProceedRight). When a deterministic instruction is at the top of the proceed stack, these new instructions have the same semantics as `proceed`.

$$\text{ProceedLeft} \quad \frac{}{(\texttt{proceedLeft} : C, X \cup S_1 \texttt{ or } S_2 : \Sigma^P) \rightarrow (S_1 : C, X \cup \Sigma^P)}$$

$$\text{ProceedRight} \quad \frac{}{(\texttt{proceedRight} : C, X \cup S_1 \texttt{ or } S_2 : \Sigma^P) \rightarrow (S_2 : C, X \cup \Sigma^P)}$$

The `if` statement allows to choose between these versions (left, right or standard) of `proceed` depending on a dynamic test. An advice in the selector language still executes the original matched instruction (or one of its branches) exactly once.

Example 4.12 defines a selector aspect that can be woven with non deterministic base programs in order to make it fair.

Example 4.12. *The following aspect balances the computation of* `serve` *for two users. It uses an integer variable* `u` *to count the difference of number of* `serve` *for* `user1` *and* `user2`*. The aspect ensures that the difference is never exceeds* 5.

```
var u := 0 : around(serve("User1") or serve("User2"))
{if(−5 < u < 5) then u−−;proceedLeft or u++;proceedRight
 else if(u ≥ 5) then u−−;proceedLeft else u++;proceedRight}
```

4.4.3 A weak intruder language

In this section, we define a specialized aspect language to manage failures. The idea is to save the state at some no-deterministic choices (using a `proceedcommit` instruction) so that in case of a failure of the chosen choice (detected by the *fail* pointcut) the program can go back to the saved state and try the other choice (using a `rollback` instruction).

We first introduce an auxiliary observer language in order to save pending branches for the non deterministic `or` instruction. The Grammar 4.13 modifies the observers grammar (Grammar 4.5) by replacing patterns P by a pattern whose static part is S_1^p `or` S_2^p and the instruction `proceed` is replaced by a new instruction `proceedCommit`. The syntax of declarations and statements remain the same. This restricted observer language is dedicated to failure management. We could have chosen a more general language by just extending the standard observer grammar with the new pointcut and instruction.

Grammar 4.13.

$$
\begin{array}{lcl}
Asp^o & ::= & D^o \texttt{ around } (S_1^p \texttt{ or } S_2^p) \wedge if(B^o)\ \{S_1^o;\ \texttt{proceedCommit};\ S_2^o\} \\
D^o & ::= & \ldots \\
S^o & ::= & \ldots \mid S_1^o \texttt{ or } S_2^o
\end{array}
$$

The semantics of such aspects is defined as follows:

$$
\begin{array}{l}
[\![\texttt{around } s_1^p \texttt{ or } s_2^p \wedge if(B^o)\ s]\!] = \\
\qquad \lambda(i : C, X \cup \Sigma^P \cup \Sigma^S). \quad \begin{array}{lll} if & match^s(s_1^p \texttt{ or } s_2^p, i) = \sigma \\ then & (\overline{a} : C, X \cup \overline{i} : \Sigma^P \cup \Sigma^S) \\ else & nil \end{array} \\
\text{with } a = \sigma(\texttt{if}(B^o) \texttt{ then } s \texttt{ else proceed})
\end{array}
$$

When the static pattern S_1^p `or` S_2^p is not matched, *nil* is returned (*i.e.,* nothing is woven). When the static pattern is matched but the dynamic condition B^o is false, the command `proceed` resumes the original execution. Finally, when both the static pattern and the dynamic condition are satisfied, the advice s is executed. The advice can perform some profiling (with its advice parts S_1^o and S_2^o) and always calls the command `proceedCommit` exactly once.

This command transforms the matched instruction S_1 `or` S_2, which has been placed at the top of the proceed stack, into another non deterministic instruction (Rule ProceedCommit) that executes the command S_1 `or` S_2 and saves in the stack Σ^S the non selected branch by calling the function *commit* (Rule Commit).

$$\textsc{ProceedCommit}\quad \frac{}{\begin{array}{c}(\texttt{proceedCommit} : C, X \cup S_1 \texttt{ or } S_2 : \Sigma^P \cup \Sigma^S) \\ \rightarrow ((commit(S_2 : C, X \cup \Sigma^P \cup \Sigma^S); S_1) \texttt{ or} \\ (commit(S_1 : C, X \cup \Sigma^P \cup \Sigma^S); S_2) : C, X \cup \Sigma^P \cup \Sigma^S)\end{array}}$$

$$\textsc{Commit}\quad \frac{}{(commit(C', \Sigma') : C, X \cup \Sigma^P \cup \Sigma^S) \rightarrow (C, X \cup \Sigma^P \cup (C', \Sigma') : \Sigma^S)}$$

When a failure occurs before the end of the advice, the state stored in Σ^S is used to rollback the program execution and try the other alternative branch. Such aspects are defined by the Grammar 4.14, where the pointcut *error* denotes error events. These events, not formalized here, can be exceptions, function calls, specific values of variables, invariant violations, etc.

Grammar 4.14.

$$Asp^r ::= \quad \texttt{around } fail \texttt{ rollback}$$

The semantics of these aspects are defined as follows:

$$\begin{array}{rl}[\![\texttt{around } fail \texttt{ rollback}]\!] = & \\ \lambda(i : C, X \cup \Sigma^P \cup \Sigma^S).\ if & match^s(fail, i) = \sigma \\ then & (\texttt{rollback} : C, X \cup \Sigma^P \cup \Sigma^S) \\ else & nil\end{array}$$

The advice `rollback` executes the configuration at the top of Σ^S (Rule ROLLBACK). This configuration corresponds to the state at the previous non deterministic choice.

$$\textsc{Rollback}\quad \frac{}{(\texttt{rollback} : C, X \cup \Sigma^P \cup (C', \Sigma') : \Sigma^S) \rightarrow (C', \Sigma')}$$

When an error is matched and Σ^S is empty (*i.e.,* there is no more pending branch to try), it is considered as a global failure and the command `rollback` ends the execution.

$$\textsc{Fail}\quad \frac{}{(\texttt{rollback} : C, X \cup \Sigma^P \cup \epsilon) \rightarrow (\bullet, X \cup \Sigma^P \cup \epsilon)}$$

The command `rollback` can only execute saved branches, so the program remains in the set of accessible states of all possible executions. Hence, such aspects are weak intruders. If the Rule ROLLBACK can be considered atomic, then the language Asp^r is a confiner language.

5 Related Work

The starting point of our study is seminal work by Katz [24] that introduces the categories spectative aspects (corresponding to observers), regulative aspects (close to our aborters and regulators) and weakly invasive aspects (similar to our weak intruders). For each category, Katz indicates which standard classes of

properties (safety, liveness and invariants) are preserved. However, that study is largely informal. Categories of aspects, classes of properties and proofs are not formalized and many definitions and results are not quite precise enough. For example, the atomic propositions on states (sp) and events (ep) are not clearly distinguished. Katz states that spectative aspects preserve safety properties. Our study shows that observers preserve only safety properties involving state properties (not event properties). Katz claims that regulative aspects (aborters) preserve safety but do not preserve liveness properties. Our study confirms this claim but only when properties involve exclusively state propositions. On the other hand, we showed that they do not preserve safety properties on events and that they preserve liveness properties involving only negation of events ($\neg ep$).

Other works focus on a specific aspect category. Dantas and Walker [8] formally describe an aspect category named harmless advice. This category corresponds to our aborters. The emphasis is on analyzing when an aspect is harmless. They propose a type system to ensure that advice does not change the final values of the base program when the woven program is not aborted. Krishnamurthi *et al.* [28] focus on aspects whose advice always returns to the join point in the original base program. They propose a modular verification technique that generates conditions to verify advice in isolation for a given property to be preserved by weaving. So, each aspect must be analyzed contrary to our approach that considers categories of aspects. This work is extended by Goldman and Katz [21] for weakly invasive aspects (weak intruders).

Rinard, Salcianu, and Bugara [35] propose categories of aspects based on a informal classification of their interactions with the base program. They distinguish two classes. The first one deals with control-flow modifications: an augmentation aspect does not modify the control-flow, a narrowing aspect can skip the function matched by the pointcut, a replacement aspect can replace the matched function by another one, a combination aspect combines the matched function and the advice to generate the actual advice. The second class deals with state modifications: an independent aspect or the function it matches cannot write a variable that is read or written by the other, an observation aspect can read a variable that the matched function writes, an actuation aspect can write a variable that the matched function reads, an interference aspect can write a variable that the matched method writes. These categories help to get a better idea of the potential impact of an aspect but the preservation of properties is not considered. Augmentation-independent aspects and augmentation-observation aspects resemble observers. Other categories can arbitrarily modify the semantics of the base program.

Clifton and Leavens [4] propose two categories: observers and assistants. As ours, observers cannot modify the specification of the base program whereas assistants can. From their examples, assistants are similar to aborters. Although they rely on Hoare-logic to explain the behavior of woven programs, the categories themselves are not formalized.

Our work is based on an abstract (*i.e.,* language independent) small step semantics of woven execution. There have been many formalization of aspect languages and weaving. For example, Wand *et al.* [39]) propose a denotational semantics for a subset of AspectJ, Bruns *et al.* [1] present a formal aspect calculus μ**ABC**, and Clifton and Leavens [5] define an operational semantics for an imperative OO language. Most of existing semantics for AOP consider object oriented base programs [22, 29, 17, 23]. Some others consider functional

languages (call-by-value λ-calculus, ML, Scheme, ...) [38, 9, 32]. Our framework, the CASB (Sec. 2.1), describes weaving as independently as possible from the base and aspect language. The CASB could be applied to many different types of programming languages (object-oriented, imperative, functional, logic, assembly, ...) and aspect languages. We committed to a specific imperative base language only to illustrate the design of specialized aspect languages.

There have also been many proposals of domain specific aspect languages. For example, Videira Lopes [37] proposes two specialized languages RIDL and COOL for remote data transfer and synchronization. Mendhekar *et al.* [33] present an aspect language which makes use of a memoization primitive to optimize image processing systems. Fradet and Hong Tuan Ha [19] define an aborter-like language to prevent the denials of service such as starvation caused by resource management. However, these languages only address a specific application domain and the preservation of properties is not studied.

6 Conclusion and Future Work

In this article, we have used a language independent semantics framework to formally define several aspects categories: observers, aborters, confiners and weak intruders. Observers do not modify the control-flow and state of the base program, aborters may in addition abort executions, confiners may modify the control-flow but remain in the reachable states and weak intruders may further leave the domain of reachable states during the execution of advice. For each category, we gave a subset of LTL properties preserved by weaving for any base program and for any aspect in the related category.

The above categories have been completed and generalized for non-deterministic programs. Selectors can select of subset of execution traces among the set of all possible traces. This category includes observers but is not comparable to aborters. Regulators are selectors that may also abort the program. The corresponding class of preserved properties are expressed as subsets of CTL* properties.

We provided examples to illustrate each category of aspects. Typically, persistence, debugging, tracing, logging and profiling aspects are observers; aspects enforcing security policies are aborters; fault-tolerance or memo aspects are either confiners or weak intruders depending on their implementation. Of course, many common aspects do not belong to our categories. For example:

- Exception aspects (see *e.g.,* [30]) can be observers if they only detect and log errors or aborters if they handle error by aborting the program (*e.g.,* contract enforcement is often implemented as aborters). However, error handling can also involve returning a default value (*e.g.,* initialization error) or retrying an action (without a proper roll-back) or terminating only a portion of the trace. In these cases, completely new states can be reached and no temporal property holds in general.

- Security aspects can be observers if they just log critical events (*e.g.,* intrusion detection aspects) or aborters when they enforce a security policy. When aspects are used to implement security mechanisms, such as access control rules, they generally modify the base program semantics.

- Context passing and change monitoring (see *e.g.,* [6], [25]) are two classical examples of production aspects. They usually change the base functionality.

Program transformations (optimizations) could be regarded as semantic-preserving aspects. Since they change the algorithm (and therefore the execution trace) they do not belong to our categories. On one hand, they preserve properties on the relevant part of the final state. On the other hand, they may violate important temporal (*e.g.,* security) properties. A special result-preserving category could be introduced. However, the class (grammar) of properties preserved would be trivial (state properties on the final result). Further, it would be hard to define a generic aspect language ensuring that aspects belong to that category.

Besides the preservation of properties, our categories have other interesting features. For example, with a few additional constraints, observers are completely independent from each other and can be composed and woven in any order. The composition of aspects produces an aspect belonging to the most category involved.

We defined restricted aspect languages that ensure aspects to belong to specific categories and therefore to preserve a class of property. In particular, we have proposed a general language for observers and aborters and a domain-specific language for memo aspects (which belongs to confiners). We also presented a selector aspect language to control non determinism and a domain-specific language for rollback aspects (belonging to weak intruders). Using that language approach, the programmer does not have to prove *a posteriori* that an aspect belongs to a category. The programmer uses the specialized aspect language that ensures *a priori* that the aspect belongs to the category.

Our work suggests several research directions. First, our classes of properties should be shown to be maximal. We should prove that each class can express exactly all properties that may be preserved by the the corresponding category. The task is not trivial since maximality is not a syntactic but a semantic criterion. For example, the property $(ep \vee \neg ep) \cup \varphi'^o$ which is preserved by observers is not a property of φ^o. However, it is semantically equivalent to $true \cup \varphi'^o$ which belongs to φ^o.

Our approach focuses on the preservation of classes of properties for any aspect of a category and for any program. It could be interesting to study less general approaches to preservation by fixing either a property, an aspect or a program. For example, the class of properties preserved by observers for a specific program is likely to be much larger than φ^o. Similarly, a fixed observer is likely to preserve larger class than φ^o even for any program. Of course, we can also fix two parameters (*e.g.,* the program and the aspect). The case where the program, the aspect and the property are fixed boils down to standard static analysis or model checking of the woven program.

The expressiveness of our languages of aspects should be studied. For instance, it is likely that all observers (resp. aborters) can be defined in the observer (resp. aborter) language. Of course, our memo language is not general: it does not enable the definition of rollback aspects that can also be confiners. However, other specialized languages belonging to the confiner family, like other dynamic optimizations and fault-tolerance aspects, could be studied. Finally, these languages could be implemented an integrated into an aspect programming

workbench allowing to reason about aspect composition and the preservation of properties.

Acknowledgments

This work has been supported by the AOSD-Europe network of excellence.

References

[1] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: A minimal aspect calculus. In *CONCUR 2004*, pages 209–224. Springer-Verlag, 2004.

[2] M. Chapman, A. Vasseur, and G. Kniesel, editors. *AOSD 2006 - Industry Track Proceedings, Bonn, Germany, March 20-24, 2006*, volume IAI-TR-2006-3. Computer Science Department III, University of Bonn, 2006.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[4] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, pages 33–44, 2002.

[5] C. Clifton and G. T. Leavens. MiniMAO1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:321–374, 2006.

[6] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[7] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, 2000.

[8] D. S. Dantas and D. Walker. Harmless advice. *SIGPLAN Not.*, 41(1):383–396, 2006.

[9] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. Polyaml: a polymorphic aspect-oriented functional programming language. In *In Proc. of ICFP'05*, pages 306–319. ACM Press, 2005.

[10] S. D. Djoko, R. Douence, and P. Fradet. Specialized aspect languages preserving classes of properties. In *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 227–236, Washington, DC, USA, 2008. IEEE Computer Society.

[11] S. Djoko Djoko. *Programmation par aspects et préservation de propriétés.* PhD thesis, University de Nantes, June 2009.

[12] S. Djoko Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *PEPM'08*, pages 135–145. ACM, 2008.

[13] S. Djoko Djoko, R. Douence, and P. Fradet. A common aspect semantics base and some applications. Technical Report AOSD-Europe Deliverable D135, August 2008.

[14] S. Djoko Djoko, R. Douence, P. Fradet, and D. Le Botlan. CASB: Common aspect semantics base. Technical Report AOSD-Europe Deliverable D54, Inria, August 2006.

[15] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE 2002*, volume 2487, pages 173–188. LNCS, October 2002.

[16] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of the 3rd Int. Conf. on Aspect-Oriented Software Development (AOSD'04)*. ACM, ACM Press, March 2004.

[17] R. Douence and L. Teboul. A crosscut language for control-flow. In *GPCE 2004*, volume 3286. Springer LNCS, October 2004.

[18] P. Fradet and S. Hong Tuan Ha. Network fusion. In *Proc. of Asian Symposium on Programming Languages and Systems (APLAS'04)*, pages 21–40. Springer-Verlag, LNCS, Vol. 3302, November 2004.

[19] P. Fradet and S. Hong Tuan Ha. Aspects of availability. In *Proc. of the Sixth International Conference on Generative Programming and Component Engineering (GPCE'07)*, pages 165–174. ACM, October 2007.

[20] J. Gibbons and G. Hutton. Proof Methods for Structured Corecursive Programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, Aug. 1999.

[21] M. Goldman and S. Katz. Maven: Modular aspect verification. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2007.

[22] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *ECOOP*, pages 54–73. Springer LNCS, 2003.

[23] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Sci. Comput. Program.*, 63(3):267–296, 2006.

[24] S. Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development*, 3880, 2006.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming*, June 1997.

[27] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching. In *ESOP*, pages 110–124, 2007.

[28] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, November 2004. ACM Press.

[29] R. Lämmel. A Semantical Approach to Method-Call Interception. In *AOSD 2002*, pages 41–55. ACM Press, Apr. 2002.

[30] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering (ICSE'00)*, pages 418–427. ACM, 2000.

[31] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[32] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. *SIGPLAN Not.*, 40(9):320–330, 2005.

[33] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Palo Alto, CA, USA, February 1997.

[34] F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, 1992.

[35] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 147–158, New York, NY, USA, 2004. ACM Press.

[36] A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 39–48, New York, NY, USA, 1985. ACM Press.

[37] C. Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, 1997.

[38] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP'03*, pages 127–139. ACM Press, 2003.

[39] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Trans. on Prog. Lang. and Sys.*, 26(5):890–910, 2004.

[40] G. Winskel. *The formal semantics of programming languages: an introduction*. The MIT Press, 1993.

A Proof for the observer category

This appendix presents in some details the proof of Theorem 3.3. The proofs of the other preservation properties are similar.

The proof makes use of two auxiliary functions $trace_b$ and rib.

The function $trace_b$ projects woven traces on their corresponding base trace. It removes steps with an advice instruction (i_a) and projects states on their corresponding base program state (Σ^b).

$$\begin{array}{l} trace_b :: Traces_{\mathcal{W}} \to Traces_{\mathcal{B}} \\ trace_b(i_b, \Sigma) : S \;=\; (i_b, \Sigma^b) : trace_b\ S \\ trace_b(i_a, \Sigma) : S \;=\; trace_b\ S \end{array}$$

The function $rib\ \alpha\ i$ returns the rank of the ith base instruction in the woven trace $\tilde{\alpha}$. If n advice instructions have been introduced/executed before reaching the ith base instructions then $rib\ \tilde{\alpha}\ i = i + n$. We use the notation $\tilde{i}$ for $rib\ \tilde{\alpha}\ i$.

The proof of theorem 3.3 relies on the following property which states that the execution trace woven with an observer can be projected (using $trace_b$) on the corresponding base execution trace.

Property A.1.

$$\begin{array}{l} \forall (C, \Sigma).\ \ \Sigma^\psi \in \mathcal{A}_o \ \Rightarrow\ trace_b(\tilde{\alpha}) = \alpha \\ with\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma) \end{array}$$

Proof. By definition

$$\begin{array}{lll} \forall (C, \Sigma).\ \ \Sigma^\psi \in \mathcal{A}_o & \Leftrightarrow & proj_b(\alpha) = proj_b(\tilde{\alpha}) \\ & & \wedge\ preserve_b(\tilde{\alpha}) \end{array}$$

The equality of traces using $proj_b$ ensures that all advice terminates whereas $preserve_b(\tilde{\alpha})$ ensures that the base store does not change during advice. So $trace_b$, which removes advice steps, the aspect and its local store, projects the woven trace on the original trace. □

When a woven execution trace can be projected on a base execution trace, the ith step of the base trace corresponds to the $\tilde{i}$th step of the woven trace.

Lemma A.2.

$$\begin{array}{l} trace_b(\tilde{\alpha}) = \alpha\ \Rightarrow \\ \forall (j \geq 1).\ \alpha_j = (i_b, \Sigma^b) \Leftrightarrow \tilde{\alpha}_{\tilde{j}} = (i_b, \Sigma) \end{array}$$

The proof is trivial using the definition of rib and $trace_b$. The following lemma is also useful.

Lemma A.3.

$$\begin{array}{l} trace_b(\tilde{\alpha}) = \alpha\ \Rightarrow \\ \quad \forall (i \geq 1).\ \forall (\widetilde{i-1} < j \leq \tilde{i}).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{i\rightarrow} \end{array}$$

The lemma states that for any base and woven trace related by projection ($trace_b$), any subtrace of the base (resp. woven) execution corresponds to a subtrace of the woven (resp. base) execution.

Theorem 3.3 is shown by proving the more general property

$$\begin{array}{ll} \Sigma^{\psi} \in \mathcal{A}_o \wedge trace_b(\tilde{\alpha}) = \alpha \;\Rightarrow & \forall (p \in \varphi^o).\alpha \models p \;\Rightarrow\; \tilde{\alpha} \models p \\ \text{with} \;\; \tilde{\alpha}_1 = (x, \Sigma) & \wedge\; \forall (p' \in \varphi'^o).\; \forall (j \geq 1). \\ & \alpha_{j\rightarrow} \models p' \;\Rightarrow\; \tilde{\alpha}_{\tilde{j}\rightarrow} \models p' \end{array}$$

When a woven trace can be projected on a base trace and the initial aspect is an observer then two properties follow. The first one corresponds to Theorem 3.3 whereas the second concerns properties of φ'^o that occur in formulae of the form $true \cup \varphi'^o$. For such a property p', all subtraces satisfying p' have their corresponding woven subtraces satisfying p'. It is easy to check that this more general property implies Theorem 3.3.

Proof. By structural induction on the formulae of φ^o and φ'^o.

Base cases

- $p = sp \quad \in \varphi^o$

$$\begin{array}{l} \alpha \models sp \;\Rightarrow\; \alpha_1 \models sp \\ \Leftrightarrow l(\Sigma_1^b, sp) = true \;\; \text{with} \;\; \alpha_1 = (i_1, \Sigma_1^b) \end{array}$$

$$trace_b(\tilde{\alpha}) = \alpha \;\Rightarrow\; \tilde{\alpha}_{\tilde{1}} = (i_1, \Sigma_1) \quad \text{by Lemma } A.2$$

Note $\tilde{\alpha}_{\tilde{1}}$ may not be the first state of the woven trace. It is only the first state with a base instruction.

Since $\Sigma^{\psi} \in \mathcal{A}_o$, the very first state of the woven trace $\tilde{\alpha}_1 = (i'_1, \Sigma'_1)$ is such that $\Sigma'^b_1 = \Sigma^b_1$ (the base state cannot be modified by a before advice) and, since state properties are only about Σ^b, then

$$\begin{array}{ll} l(\Sigma_1^b, sp) & \Rightarrow\; l(\Sigma'_1, sp) \\ & \Rightarrow\; \tilde{\alpha}_1 \models sp \\ & \Rightarrow\; \tilde{\alpha} \models sp \end{array}$$

- $p = ep \quad \in \varphi'^o$

$$\begin{array}{l} \forall (j \geq 1).\;\; \alpha_{j\rightarrow} \models ep \;\Rightarrow\; \alpha_j \models ep \;\Rightarrow\; m(i_j, ep) \\ \text{By Lemma } A.2 \\ \alpha_j = (i_j, \Sigma^b) \;\Rightarrow\; \tilde{\alpha}_{\tilde{j}} = (i_j, \Sigma) \\ \text{so} \quad m(i_j, ep) = m(\tilde{\alpha}_{\tilde{j}}, ep) \\ \text{and } \tilde{\alpha}_{\tilde{j}} \models ep \\ \text{and therefore} \;\; \tilde{\alpha}_{\tilde{j}\rightarrow} \models ep \end{array}$$

- $p = \neg sp \in \ \varphi^o$ and $p = \neg ep, sp, \neg sp \ \in \ \varphi'^o$ are similar to the previous cases.

Induction

For any subformula δ of φ^o the induction hypothesis is:

$$\alpha \models \delta \ \Rightarrow \ \tilde{\alpha} \models \delta$$

and for any subformula δ of φ'^o:

$$\forall(j \geq 1).\ \alpha_{j\rightarrow} \models \delta \ \Rightarrow \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \delta$$

To apply the hypothesis we just check that the corresponding traces are in relation (*i.e.*, $trace_b(\tilde{\alpha}) = \alpha$). We do not check the second condition (the current aspect is an observer). It is easy to verify that a trace with an initial observer aspect has only observers throughout.

- $p = \varphi_1^o \ \wedge \ \varphi_2^o \ \in \varphi^o$

$$\begin{array}{l}\alpha \models \varphi_1^o \ \wedge \ \varphi_2^o \\ \Rightarrow \alpha \models \varphi_1^o \ \wedge \ \alpha \models \varphi_2^o \\ \Rightarrow \tilde{\alpha} \models \varphi_1^o \ \wedge \ \tilde{\alpha} \models \varphi_2^o \quad \text{by induction hypothesis} \\ \Rightarrow \tilde{\alpha} \models \varphi_1^o \ \wedge \ \varphi_2^o\end{array}$$

- Similarly for $p = \varphi_1^o \ \vee \ \varphi_2^o \ \in \varphi^o$

- $p = \varphi_1^o \cup \varphi_2^o \ \in \varphi^o$

$$\alpha \models \varphi_1^o \cup \varphi_2^o \ \Rightarrow \ \exists(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_2^o \wedge \forall(1 \leq i < j).\ \alpha_{i\rightarrow} \models \varphi_1^o$$

By lemma $A.3$
$trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow \ trace_b(\tilde{\alpha}_{\widetilde{j-1}+1\rightarrow}) = \alpha_{j\rightarrow}$
$\Rightarrow \ \tilde{\alpha}_{\widetilde{j-1}+1\rightarrow} \models \varphi_2^o$ by induction hypothesis
$\Rightarrow \ \exists(k \geq 1).\tilde{\alpha}_{k\rightarrow} \models \varphi_2^o$ with $k = \widetilde{j-1}+1$

$\forall(1 \leq l < k).\ \exists(1 \leq i < j).$
$k = \widetilde{j-1}+1 \ \wedge \ \widetilde{i-1} < l \leq \tilde{i}$
so $trace_b(\tilde{\alpha}_{l\rightarrow}) = \alpha_{i\rightarrow}$ by Lemma $A.3$
and since $\alpha_{i\rightarrow} \models \varphi_1^o$ for all such i
$\tilde{\alpha}_{l\rightarrow} \models \varphi_1^o$ by induction hypothesis

Thus $\tilde{\alpha} \models \varphi_1^o \cup \varphi_2^o$

- $p = true \cup \varphi'^o \quad \in \varphi^o$

$$\alpha \models true \cup \varphi'^o \quad \Rightarrow \quad \exists(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi'^o \wedge \forall(1 \leq i < j).\ \alpha_{i\rightarrow} \models true$$

by induction hypothesis, we get

$$\tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi'^o$$

so, taking $k = \tilde{j}$, we have $(\exists k \geq 1).\ \tilde{\alpha}_{k\rightarrow} \models \varphi'^o$
and since trivially $\forall(1 \leq l < \tilde{j}).\ \tilde{\alpha}_{l\rightarrow} \models true$
we have

$$\tilde{\alpha} \models true \cup \varphi'^o$$

- Similarly for $p = \varphi_1^o W \varphi_2^o \quad \in \varphi^o$

- $p = \varphi_1'^o \ \wedge \ \varphi_2'^o \quad \in \varphi'^o$

$$\begin{array}{l} \forall(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_1'^o \ \wedge \ \varphi_2'^o \\ \Rightarrow \ \alpha_{j\rightarrow} \models \varphi_1'^o \ \wedge \ \alpha_{j\rightarrow} \models \varphi_2'^o \\ \Rightarrow \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_1'^o \ \wedge \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_2'^o \quad \text{by induction hypothesis} \\ \Rightarrow \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_1'^o \ \wedge \ \varphi_2'^o \end{array}$$

- Similarly for $p = \varphi_1'^o \ \vee \ \varphi_2'^o \quad \in \varphi'^o$
- $p = \varphi_1^o \cup \varphi_2^o \quad \in \varphi'^o$

$$\forall(j \geq 1).\ \alpha_{j\rightarrow} \models \varphi_1^o \cup \varphi_2^o \quad \Rightarrow \quad \exists(k \geq j).\ \alpha_{k\rightarrow} \models \varphi_2^o \wedge \forall(j \leq l < k).\ \alpha_{l\rightarrow} \models \varphi_1^o$$

By lemma $A.3$
$trace_b(\tilde{\alpha}) = \alpha \Rightarrow \ trace_b(\tilde{\alpha}_{\widetilde{k-1}+1\rightarrow}) = \alpha_{k\rightarrow}$
$\Rightarrow \ \tilde{\alpha}_{\widetilde{k-1}+1\rightarrow} \models \varphi_2^o$ by induction hypothesis
$\Rightarrow \ \exists(m \geq \tilde{j} \geq j).\tilde{\alpha}_{m\rightarrow} \models \varphi_2^o$ taking $m = \widetilde{k-1}+1$

$\forall(j \leq n < m).\ \exists(j \leq l < k).$
$m = \widetilde{k-1}+1 \ \wedge \ \widetilde{l-1} < n \leq \tilde{l}$
so $trace_b(\tilde{\alpha}_{n\rightarrow}) = \alpha_{l\rightarrow}$ by Lemma $A.3$
and since $\alpha_{l\rightarrow} \models \varphi_1^o$ for all such l
$\Rightarrow \ \tilde{\alpha}_{n\rightarrow} \models \varphi_1^o$ by induction hypothesis

Thus $\exists(m \geq \tilde{j} \geq j).\tilde{\alpha}_{m\rightarrow} \models \varphi_2^o$
$\wedge \quad \forall(j \leq \tilde{j} \leq n < m).\tilde{\alpha}_{n\rightarrow} \models \varphi_1^o$

and therefore $\tilde{\alpha}_{\tilde{j}\rightarrow} \models \varphi_1^o \cup \varphi_2^o$

- $p = true \cup \varphi'^o \quad \in \varphi'^o$

  $$\begin{array}{ll} \forall(j \geq 1). & \alpha_{j\rightarrow} \models true \cup \varphi'^o \\ \Rightarrow & \exists(k \geq j).\ \alpha_{k\rightarrow} \models \varphi'^o \wedge \\ & \forall(j \leq i < k).\ \alpha_{i\rightarrow} \models true \end{array}$$

  by induction hypothesis, we get

  $$\tilde{\alpha}_{\tilde{k}\rightarrow} \models \varphi'^o$$

  so $k \geq j \ \Rightarrow\ \tilde{k} \geq \tilde{j}$ and since trivially

  $$\forall(\tilde{j} \leq l < \tilde{k}).\ \tilde{\alpha}_{l\rightarrow} \models true$$

  then

  $$\tilde{\alpha}_{\tilde{j}\rightarrow} \models true \cup \varphi'^o$$

- Similarly for $p = \varphi_1^o W \varphi_2^o \quad \in \varphi'^o$

□

B Semantics of Prog

This appendix provides the semantics of the base language introduced in Section 4.1. That simple imperative language is very standard and so is its semantics. However, we present it to illustrate how the semantics of a realistic language can respect the form imposed by the common semantic base (Section 2.1)

The semantics of expressions is given in a denotational style. The semantics of declarations (D) and statements (S) are given by a small-step structural operational semantics.

B.1 Expressions

The semantics of arithmetic expressions is given by the function $\mathcal{E}_a$ that takes a syntactic expression A, the states of global and local variables represented by the functions Σ^b_g and Σ^b_g and returns an integer.

$$\begin{array}{rcl}
\mathcal{E}_a[\![n]\!]\ \Sigma^b_g\ \Sigma^b_{l1} & = & \mathcal{N}[\![n]\!] \\
\mathcal{E}_a[\![g]\!]\ \Sigma^b_g\ \Sigma^b_{l1} & = & \Sigma^b_g(g) \\
\mathcal{E}_a[\![l]\!]\ \Sigma^b_g\ \Sigma^b_{l1} & = & \Sigma^b_{l1}(l) \\
\mathcal{E}_a[\![A_1{+}A_2]\!]\ \Sigma^b_g\ \Sigma^b_{l1} & = & (\mathcal{E}_a[\![A_1]\!]\ \Sigma^b_g\ \Sigma^b_{l1}) + (\mathcal{E}_a[\![A_2]\!]\ \Sigma^b_g\ \Sigma^b_{l1})
\end{array}$$

The function $\mathcal{N}$ takes a syntactic integer and returns the corresponding mathematical integer in $\mathbb{Z}$.

The semantics of boolean expressions is given by a function $\mathcal{E}_b$ that takes an expression B, the functions Σ^b_g and Σ^b_g (used to evaluate the arithmetic expressions) and returns a boolean in $\mathbf{Bool} = \{\mathrm{tt}, \mathrm{ff}\}$. This function is very similar to $\mathcal{E}_a$ and we do not describe it here.

B.2 Declarations

The operational treatment of declarations produces two environments:

- Σ^b_g records the value of global variables and is read and written by assignments;
- Σ^b_{proc} is used by call statements to fetch the name of parameters and the body of the procedure.

$$\textsc{DVar}\quad \frac{\mathcal{E}_a[\![A]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \nu}{(\texttt{var}\ g{:=}A, \Sigma^b_g, \Sigma^b_{proc}) \rightarrow_d (\Sigma^b_g[g \mapsto \nu], \Sigma^b_{proc})}$$

$$\textsc{DProc}\quad \frac{}{(\texttt{proc}\ I(l_1, \ldots l_n)\ S, \Sigma^b_g, \Sigma^b_{proc}) \rightarrow_d (\Sigma^b_g, \Sigma^b_{proc}[I \mapsto ((l_1, \ldots, l_n), S)])}$$

$$\textsc{DSeq}_1\quad \frac{(D_1, \Sigma^b_g, \Sigma^b_{proc}) \rightarrow_d (D'_1, \Sigma'^b_g, \Sigma'^b_{proc})}{(D_1;D_2, \Sigma^b_g, \Sigma^b_{proc}) \rightarrow_d (D'_1;D_2, \Sigma'^b_g, \Sigma'^b_{proc})}$$

$$\textsc{DSeq}_2 \quad \frac{(D_1, \Sigma_g^b, \Sigma_{proc}^b) \rightarrow_d (\Sigma_g'^b, \Sigma_{proc}'^b)}{(D_1;D_2, \Sigma_g^b, \Sigma_{proc}^b) \rightarrow_d (D_2, \Sigma_g'^b, \Sigma_{proc}'^b)}$$

B.3 Statements

The semantics of statements is given by the relation $\rightarrow_b$ used in many definitions and proofs of this article. A configuration (C, Σ) is made of the code and a store made of two functions Σ_g^b and Σ_l^b representing the stores for global and local (*i.e.,* parameters) variables respectively. The initial Σ_g^b depends on the declarations of global variables and is computed by the semantic relation $(\rightarrow_d)$. The initial Σ_l^b is a stack with an empty context (*i.e.,* associating $\perp$ to any local variable). Each time a procedure is called, a new context associating values to parameters is pushed to Σ_l^b. Each time a procedure returns, a context is popped. The semantics also uses Σ_{proc}^b (computed by $\rightarrow_d$) for calls. This environment is left implicit in configurations since it is only read and never modified.

In the following, the operator ":" is supposed associative and programs are supposed to in the form $(i_1 : i_2 : \ldots : \Sigma_l^b)$. Writing an expression such as $S : C$ may involve implicit applications of associativity rule to get the previous linear form.

The `skip` instruction leaves the store unchanged and the continuation is executed.

$$\textsc{Skip} \quad \frac{}{(\texttt{skip} : C, (\Sigma_g^b, \Sigma_{l1}^b)) \rightarrow_b (\Sigma_g^b, \Sigma_l^b))}$$

The `abort` instruction terminates the programs: the current instruction becomes the final instruction, the stack of local variables is flushed, the global variables stay unchanged.

$$\textsc{Abort} \quad \frac{}{(\texttt{abort} : C, (\Sigma_g^b, \Sigma_l^b)) \rightarrow_b (\epsilon : \bullet, (\Sigma_g^b, \perp : \epsilon))}$$

The final instruction just keeps looping leaving global variables unchanged. The stack of local variables must be empty since all procedures have returned (or the stack has been flushed by an `abort` instruction.

$$\textsc{Final} \quad \frac{}{(\epsilon : \bullet, (\Sigma_g^b, \Sigma_l^b)) \rightarrow_b (\epsilon : \bullet, (\Sigma_g^b, \perp : \epsilon))}$$

The assignment instruction is specified by two rules depending whether the assigned variable is local or global. The assignment takes place in Σ_g^b or in the first context Σ_{l1}^b of the stack Σ_l^b.

$$\textsc{Set}_1 \quad \frac{\mathcal{E}_a[\![A]\!]\ \Sigma_g^b\ \Sigma_{l1}^b = \nu}{(g{:=}A : C, (\Sigma_g^b, \Sigma_{l1}^b : \Sigma_{ls}^b)) \rightarrow_b (C, (\Sigma_g^b[g \mapsto \nu], \Sigma_{l1}^b : \Sigma_{ls}^b))}$$

$$\textsc{Set}_2 \quad \frac{\mathcal{E}_a[\![A]\!]\ \Sigma_g^b\ \Sigma_{l1}^b = \nu}{(l{:=}A : C, (\Sigma_g^b, \Sigma_{l1}^b : \Sigma_{ls}^b)) \rightarrow_b (C, (\Sigma_g^b \cup \Sigma_{l1}^b[l \mapsto \nu] : \Sigma_{ls}^b))}$$

A procedure call involves fetching the formal parameters and body in the environment Σ_{proc}^b. The actual parameters are evaluated and a new context (associating formal parameters to their value) is pushed onto the stack. The body of

the procedure followed by `return` is placed before the continuation C.

$$\textsc{Call}\quad \frac{\Sigma^b_{proc}(p) = ((l_1,\ldots,l_n),S) \qquad \mathcal{E}_a[\![A_1]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \nu_1,\ldots,\mathcal{E}_a[\![A_n]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \nu_n}{\begin{array}{c}(p(A_1,\ldots,A_n) : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b \\ (S : \texttt{return} : C, (\Sigma^b_g, \{l_1 \mapsto \nu_1,\ldots,l_n \mapsto \nu_n\} : \Sigma^b_{l1} : \Sigma^b_{ls}))\end{array}}$$

The special instruction `return` marks the end of a procedure evaluation. It pops the context before proceeding to the continuation.

$$\textsc{Return}\quad \frac{}{(\texttt{return} : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (C, (\Sigma^b_g, \Sigma^b_{ls}))}$$

The sequencing is formalized by linearizing the statements in the code component.

$$\textsc{Seq}\quad \frac{}{(S_1; S_2 : C, (\Sigma^b_g, \Sigma^b_l)) \rightarrow_b (S_1 : S_2 : C, (\Sigma^b_g, \Sigma^b_l)}$$

The rules for conditional are standard.

$$\textsc{If}_1\quad \frac{\mathcal{E}_b[\![B]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \mathrm{tt}}{(\texttt{if}(B)\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (S_1 : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls}))}$$

$$\textsc{If}_2\quad \frac{\mathcal{E}_b[\![B]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \mathrm{ff}}{(\texttt{if}(B)\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (S_2 : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls}))}$$

While loops are evaluated by duplicating their body until the condition is false.

$$\textsc{While}_1\quad \frac{\mathcal{E}_b[\![B]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \mathrm{tt}}{(\texttt{while}(B)\ S : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (S : \texttt{while}(B)\ S : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls}))}$$

$$\textsc{While}_2\quad \frac{\mathcal{E}_b[\![B]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = \mathrm{ff}}{(\texttt{while}(B)\ S : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls}))}$$

Loops are evaluated by replicating their body the number of times specified by their arithmetic expression.

$$\textsc{Loop}_1\quad \frac{\mathcal{E}_a[\![A]\!]\ \Sigma^b_g\ \Sigma^b_{l1} = n\ \wedge\ n \geq 1}{(\texttt{loop}(A)\ S : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (\underbrace{S : \ldots : S}_{n\ times} : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls}))}$$

If this number is less or equal to zero, it amounts to skip to the next instruction.

$$\textsc{Loop}_2\quad \frac{\mathcal{E}_a[\![A]\!]\ \Sigma^b_g\ \Sigma^b_{l1} \leq 0}{(\texttt{loop}(A)\ S : C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls})) \rightarrow_b (C, (\Sigma^b_g, \Sigma^b_{l1} : \Sigma^b_{ls}))}$$

C Proof for the observer language

This appendix presents the proof of property 4.7. It relies on Property C.1 which implies directly Property 4.7 by definition of $\mathcal{A}_o$. The proofs of others properties are similar.

Property C.1.

$$\begin{array}{l}\forall(a \in Asp^o).\forall(C,\Sigma).\ \ \Sigma^\psi = [\![a]\!] \\ \qquad \Rightarrow\ proj_b(\alpha) = proj_b(\tilde{\alpha})\ \wedge\ preserve_b(\tilde{\alpha}) \\ with\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)\end{array}$$

Property C.1 is proved using Lemmas C.2 and C.5 which show respectively that aspects do not modify the base program state and its control flow.

In proofs, if α is a trace then its i^{th} element is denoted by α_i and its prefix $\alpha_1 : \ldots : \alpha_j$ by $\alpha_{\rightarrow j}$. The auxiliary functions $proj_b$ and $preserve_b$ are defined as follows:

$$\begin{array}{l}proj_b : Traces_{\mathcal{B}} \cup Traces_{\mathcal{W}} \rightarrow Sequence_{i_b} \\ proj_b((i_b,\Sigma) : T)\ =\ i_b : (proj_b\ T) \\ proj_b((i_a,\Sigma) : T)\ =\ proj_b\ T\end{array}$$

$$\begin{array}{lcl}preserve_b : Traces_{\mathcal{W}} \rightarrow bool & & \\ preserve_b(\tilde{\alpha}) & = & \forall(j \geq 1).\ \tilde{\alpha}_j = (i_a,\Sigma_j) \\ & & \Rightarrow\ \tilde{\alpha}_{j+1} = (i,\Sigma_{j+1})\ \wedge\ \Sigma^b_j = \Sigma^b_{j+1}\end{array}$$

where $Traces_{\mathcal{B}}$, $Traces_{\mathcal{W}}$ and $Sequence_{i_b}$ denote the sets of base program execution traces, woven execution traces and sequences of base instructions respectively.

Lemma C.2.

$$\begin{array}{l}\forall(a \in Asp^o).\forall(C,\Sigma).\ \ \Sigma^\psi = [\![a]\!]\ \Rightarrow\ preserve_b(\tilde{\alpha}) \\ with\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)\end{array}$$

Proof. It is easy to see (proof by cases) that all i_a instructions of $\{S^o; \mathsf{proceed}; S^o\}$ modify only Σ^a after reduction by $\rightarrow$. Indeed, instructions of S^o write only aspects variables and the proceed stack Σ^P (modified by proceed) is a subset of Σ^a ($\Sigma^P \subset \Sigma^a$). □

To prove Lemma C.5, we first prove Lemma C.3 which expresses that for any prefix of α, there exists a prefix of $\tilde{\alpha}$ equal after projection on base program instructions.

Lemma C.3.

$$\begin{array}{l}\forall(a \in Asp^o).\forall(C,\Sigma).\ \ \Sigma^\psi = [\![a]\!] \\ \quad \Rightarrow\ \forall(l \geq 1).\exists(m \geq l).\ \ proj_b(\alpha_{\rightarrow l}) = proj_b(\tilde{\alpha}_{\rightarrow m}) \\ with\ \ \alpha = \mathcal{B}(C,\Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C,\Sigma)\end{array}$$

Proof. By induction on length of α and $\tilde{\alpha}$ and assuming that the advice terminates (Hypothesis C.4).

Hypothesis C.4.

$$\forall(D^o\ \texttt{around}\ P\ \{s\} \in Asp^o).\ \ s\ \ terminates$$

By Hypothesis C.4
$(\forall(j \geq 1).\ \tilde{\alpha}_j = (i_a, \_)\ \Rightarrow\ \exists(k > j).\ \tilde{\alpha}_k = (i_b, \_))$

Base case $l = 1$

$\alpha_{\to 1} = (i_1, \_)$

$\Sigma^{\psi}(i_1 : \_, \_) = nil\ \Rightarrow\ \tilde{\alpha}_{\to 1} = (i_1, \_)$
by definition of $\mathcal{W}(C, \Sigma)$
$\Rightarrow\ proj_b(\alpha_{\to 1}) = proj_b(\tilde{\alpha}_{\to 1})$
by definition of $proj_b$

$\Sigma^{\psi}(i_1 : \_\ , \_\ ) \neq nil\ \Rightarrow\ \tilde{\alpha}_{\to 1} = (i_a, \_\ )$
by definition of $\mathcal{W}(C, \Sigma)$
$\Rightarrow\ \exists(m > 1).\ \tilde{\alpha}_m = (i_1, \_\ ) \wedge$
$\forall(m' < m).\ \tilde{\alpha}_{m'} = (i_a, \_\ )$
by Hypothesis $C.4$, and definition of $\mathcal{W}(C, \Sigma)$
$\Rightarrow \exists(m > 1).\ proj_b(\alpha_{\to 1}) = proj_b(\tilde{\alpha}_{\to m})$
by definition of $proj_b$

Induction $l = n$

We assume that

$$\exists(m \geq n).\ proj_b(\alpha_{\to n}) = proj_b(\tilde{\alpha}_{\to m})$$

and show that this is the case for $l = n + 1$
$\alpha_{\to n+1} = \alpha_1 : \ldots : \alpha_n : \alpha_{n+1}\ \wedge\ \alpha_{n+1} = (i_{n+1}, \_\ )$

$\Sigma^{\psi}(i_{n+1} : \_\ , \_\ ) = nil$
$\Rightarrow\ \exists(m' = m + 1 \geq n + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_\ )$
$\vee\ (\exists(m' > m + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_\ )$
$\wedge\ \forall(m < m'' < m').\ \tilde{\alpha}_{m''} = (i_a, \_\ ))$
by Hypothesis $C.4$, and definition of $\mathcal{W}(C, \Sigma)$
$\Rightarrow\ \exists(m' \geq n + 1).\ proj_b(\alpha_{\to n+1}) = proj_b(\tilde{\alpha}_{\to m'})$
by definition of $proj_b$

$\Sigma^{\psi}(i_{n+1} : \_\ , \_\ ) \neq nil$
$\Rightarrow\ \exists(m' > m + 1).\ \tilde{\alpha}_{m'} = (i_{n+1}, \_\ )$
$\wedge\ \forall(m < m'' < m').\ \tilde{\alpha}_{m''} = (i_a, \_\ ))$
by Hypothesis $C.4$, and definition of $\mathcal{W}(C, \Sigma)$
$\Rightarrow\ \exists(m' > n + 1).\ proj_b(\alpha_{\to n+1}) = proj_b(\tilde{\alpha}_{\to m'})$
by definition of $proj_b$ and $\mathcal{W}(C, \Sigma)$ □

Lemma C.5.

$$\begin{array}{l}\forall(a \in Asp^o).\forall(C, \Sigma).\\ \Sigma^{\psi} = [\![a]\!]\ \Rightarrow\ proj_b(\alpha) = proj_b(\tilde{\alpha})\\ with\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ and\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)\end{array}$$

Proof. Using Lemma C.3 and the coinduction relation [20] below

$$\begin{array}{l}proj_b(\alpha) = proj_b(\tilde{\alpha})\\ \Leftrightarrow\ \forall(k \geq 1).\ \ approx\ k\ proj_b(\alpha)\ \ =\ \ approx\ k\ proj_b(\tilde{\alpha})\end{array}$$

where *approx* k α is a function returning the k-first elements of the sequence α. □

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399