Aspects Preserving Properties

Simplice Djoko Djoko
INRIA, EMN, LINA
INRIA Grenoble - Rhône-Alpes
655, av. de l'Europe
38330 Montbonnot, France
simplice.djokodjoko@inria.fr

Rémi Douence
EMN, INRIA, LINA
École des Mines de Nantes
4, rue Alfred Kastler - BP 20722
44307 Nantes Cedex 3, France
douence@emn.fr

Pascal Fradet
INRIA
INRIA Grenoble - Rhône-Alpes
655, av. de l'Europe
38330 Montbonnot, France
Pascal.Fradet@inria.fr

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. It is then sufficient to check that an aspect belongs to a specific category to know which properties will remain satisfied by woven programs.

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 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 give examples to illustrate each category and prove the preservation of one class of properties by one category of aspects in the appendix.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; F.3.2 [*Semantics of Programming Languages*]; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Languages, Verification

Keywords Aspect weaving, semantics, temporal properties, proofs

1. Introduction

Aspect oriented programming (AOP) proposes to modularize concerns that crosscut the base program [13]. 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 [11], 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 [16] for deterministic programs and CTL* [2] 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.

The article starts by considering deterministic programs. Section 2 introduces the formal framework used in the rest of the paper. Section 3 recalls the syntax and semantics of linear temporal logic (LTL). We then define the categories of aspects and their corresponding classes of LTL properties: observers (Section 4.1),

PEPM'08, January 7–8, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-977-7/08/0001…$5.00

aborters (Section 4.2), confiners (Section 4.3) and weak intruders (Section 4.4). Section 5 sketches the corresponding study in a nondeterminism setting. It presents the semantics of nondeterministic programs, the computation temporal logic (CTL*) and two new categories of aspects (selectors and regulators) and their corresponding classes of CTL* properties. Section 6 reviews some related work and Section 7 discusses possible future research directions and concludes. The appendix presents the proof of preservation for observers.

2. Semantic 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 [7]. That abstract framework applies to most base and aspect languages. We define execution traces of base and woven programs and we show they are related. We focus in this section on deterministic programs. The study of non-determinism is postponed to section 5.

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 interested reader will find in [7] the semantic descriptions of a small arithmetic language, imperative language, and 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 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 wether 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:

$$\text{REDUCE} \quad \frac{(C, \Sigma) \rightarrow_b (C', \Sigma') \quad w(C', \Sigma') = (C'', \Sigma'')}{(C, \Sigma) \rightarrow (C'', \Sigma'')}$$

A reduction step $\rightarrow$ of the woven program first reduces the first instruction of the current configuration using $\rightarrow_b$, then it 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

$$\text{WEAVE0} \quad \frac{\Sigma^\psi(C, \Sigma) = nil}{w(C, \Sigma) = (C, \Sigma)}$$

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

$$\text{WEAVE1} \quad \frac{\Sigma^\psi(C, \Sigma) = (C', \Sigma') \quad w(C', \Sigma') = (C'', \Sigma'')}{w(C, \Sigma) = (C'', \Sigma'')}$$

  where
  - C' is the new code in which an advice is inserted before, after or around the current instruction of C (see [7] 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. We assume that weaving only depends on the current instruction (not on the continuation). The interested reader will find in [7] 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 $\rightarrow_b$ 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):

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

So, a base program always starts by the reduction step

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

whereas a woven execution starts by the reduction step

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

2.2 Base and Woven Execution Traces

In the following, programs are represented by their execution traces. Programs are supposed to end 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 1).

DEFINITION **1.**

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

Since properties concern only states and current instructions, continuation (the control stack) does not appear in traces. We write $\mathcal{W}(C_0, \Sigma_0)$ for the infinite woven execution trace (definition 2).

DEFINITION **2.**

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

Note that in both definitions, the initial instruction i_0 (*i.e.,* $start$) does not appear.

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.

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

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.

$$\begin{array}{rcl} \multicolumn{3}{l}{preserve_b : Traces_{\mathcal{W}} \rightarrow bool} \\ preserve_b(\tilde{\alpha}) & = & \forall (j \geq 1).\ \tilde{\alpha}_j = (i_a, \Sigma_j) \\ & & \Rightarrow\ \tilde{\alpha}_{j+1} = (i, \Sigma_{j+1})\ \wedge\ \Sigma^b_j = \Sigma^b_{j+1} \end{array}$$

These functions are used in the definition of aspect categories.

3. Linear Temporal Logic

Linear Temporal Logic (LTL) permits to define a wide range of properties of execution traces [16]. In this section, we define the syntax and semantics of LTL formulae *w.r.t.* our (base and woven) execution traces. We review standard classes of temporal properties and briefly discuss why these classes are not, in general, preserved by weaving.

3.1 Atomic propositions

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

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 $m\ ::\ Instruction \times Ep \rightarrow bool$, 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:

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

- 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 parameters. The function l is overloaded in order to take a trace element as parameter:

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

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

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

3.2 Semantics of LTL

We consider LTL formulae in positive normal form *i.e.,* where negation occurs only on atomic propositions (Grammar 1). In φ, the operator $\bigcirc$ is read "next", $\cup$ is read "until", and W is read "weak until".

GRAMMAR **1.**

$$\begin{array}{rcl} \varphi & ::= & ap\ \mid\ \neg ap\ \mid\ \varphi_1 \vee \varphi_2\ \mid\ \varphi_1 \wedge \varphi_2\ \mid \\ & & \bigcirc\varphi\ \mid\ \varphi_1 \cup \varphi_2\ \mid\ \varphi_1 W \varphi_2 \end{array}$$

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.

3.3 Standard Classes of Temporal properties

Standard classes of temporal properties [18] 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 "the state of the system is eventually saved". An around aspect replacing the calls to the function `backup` by an empty advice 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.

The rest of this article is devoted to identifying categories of aspects that preserve classes of temporal properties.

4. Categories of aspects

For deterministic programs, our four aspect categories are: observers ($\mathcal{A}_o$), aborters ($\mathcal{A}_a$), confiners ($\mathcal{A}_c$) and weak intruders ($\mathcal{A}_w$). 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 $\mathcal{A}_x$. Aspect categories 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. 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. On the other hand, the advantage of our 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 or will be.

For these reasons, the classes of 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:

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

represents an execution trace where the first and the second step satisfies $x = 0$ and the third step satisfies $x = 1$ and has $print$ as its current instruction (*e.g.,* the second instruction has changed the valued of x). This trace satisfies, for example, the property $(x = 0)W print$.

4.1 Observers

An observer (Definition 3) 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.**

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

Definition 3 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 2.

GRAMMAR **2.**

$$\begin{array}{lll} \varphi^o & ::= & sp \mid \neg sp \mid \varphi_1^o \vee \varphi_2^o \mid \varphi_1^o \wedge \varphi_2^o \mid \varphi_1^o \cup \varphi_2^o \mid \\ & & \varphi_1^o W \varphi_2^o \mid true \cup \varphi'^o \\ \\ \varphi'^o & ::= & ep \mid \neg ep \mid sp \mid \neg sp \mid \varphi_1'^o \vee \varphi_2'^o \mid \varphi_1'^o \wedge \varphi_2'^o \mid \\ & & \varphi_1^o \cup \varphi_2^o \mid \varphi_1^o W \varphi_2^o \mid true \cup \varphi'^o \end{array}$$

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 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, $read W false$ (*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 $read$

$$read : write : read : read : \dots$$

the property is not satisfied any more.

The theorem 1 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 **1.**

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

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 [1], Govidranj *et al.* present a tool named InfraRED. The tool is based on observer AspectJ aspects to monitor J2EE applications and to detect and analyze performance problems.

4.2 Aborters

An aborter (Definition 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:

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

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 **4.**

$$\begin{array}{lll} \forall (C, \Sigma).\ \Sigma^\psi \in \mathcal{A}_a & \Leftrightarrow & preserve_b(\tilde{\alpha}) \wedge \\ & & proj_b(\alpha) = proj_b(\tilde{\alpha})\ \vee (\exists (i \geq 0). \\ & & \exists (j \geq i).\ proj_b(\alpha_{\rightarrow i}) = proj_b(\tilde{\alpha}_{\rightarrow j}) \\ & & \wedge\ \forall (k > j).\tilde{\alpha}_k = (\epsilon, \_)) \\ \multicolumn{3}{l}{\textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)} \end{array}$$

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) is a subset of the set of properties preserved by observers (Grammar 2).

GRAMMAR **3.**

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

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

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

$$print : print : print : read : \epsilon : \dots$$

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

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

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

$$(x = 0, abort) : (x = 0, \epsilon) : (x = 0, \epsilon) : \dots$$

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 by aborter aspects is formalized by Theorem 2.

THEOREM **2.**

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

Examples of 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 [5], 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 [9], aspects are timed constraints which may terminate programs to guarantee availability of shared resources. In [1], 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.

4.3 Confiners

An aspect is a confiner (Definition 5) 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 5 formalizes the fact that the base state of any configuration in the woven trace is reachable by the base program.

DEFINITION **5.**

$$\begin{array}{lll} \forall (C, \Sigma).\ \Sigma^\psi \in \mathcal{A}_c & \Leftrightarrow & \forall (j \geq 1).\ \tilde{\alpha}_j = (i, \Sigma_j) \\ & & \wedge\ \Sigma_j^b\ \in\ Reach_b(C, \Sigma^b) \\ \multicolumn{3}{l}{\textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)} \end{array}$$

Observers and aborters are included in the category $\mathcal{A}_c$ of confiners. The set of properties preserved by confiners (Grammar 4) is a subset of the set of properties preserved by aborter aspects (Grammar 3).

GRAMMAR **4.**

$$\varphi^c ::= sp \mid \neg sp \mid \varphi_1^c \vee \varphi_2^c \mid \varphi_1^c \wedge \varphi_2^c \mid \varphi_1^c W \mathit{false}$$

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 $\varphi_1^c W \varphi_2^c$ 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 4 by confiners is formalized by Theorem 3.

THEOREM **3.**

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

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.

4.4 Weak intruders

An aspect is a weak intruder (definition 6) 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 6 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 **6.**

$$\begin{array}{lll} \forall (C, \Sigma).\ \Sigma^{\psi} \in \mathcal{A}_w & \Leftrightarrow & \forall (j \geq 1).\ \tilde{\alpha}_j = (i_b, \Sigma_j) \\ & & \Rightarrow \Sigma_j^b \ \in\ Reach_b(C, \Sigma^b) \\ \multicolumn{3}{l}{\textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)} \end{array}$$

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 7) 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 **7.**

$$\begin{array}{lll} \forall (C, \Sigma).\ \Sigma^{\psi}\ \textit{terminates} & \Leftrightarrow & \exists (j \geq 1). \forall (k > j). \\ & & \tilde{\alpha}_k = (i_b, \Sigma_k) \\ \multicolumn{3}{l}{\textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)} \end{array}$$

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, $(x = 0 \vee x = 1) W \mathit{false}$ is eventually satisfied (*i.e.,* $\Diamond((x = 0 \vee x = 1) W \mathit{false})$).

Theorem 4 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 **4.**

$$\begin{array}{ll} \forall (C, \Sigma). & \Sigma^{\psi} \in \mathcal{A}_w \ \wedge\ \Sigma^{\psi}\ \textit{terminates} \ \Rightarrow \\ & \forall (p \in \varphi^c).\ \alpha \models p \ \Rightarrow\ \tilde{\alpha} \models \Diamond p \\ \multicolumn{2}{l}{\textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma)} \end{array}$$

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.

5. Non-Deterministic Case

In the previous section, we have presented aspect categories preserving classes of properties for deterministic programs. 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.

We first define the semantics of non-deterministic programs as set of (infinite) execution traces. The categories of aspects are defined based on this semantics and the same auxiliary functions ($proj_b$ and $preserve_b$). The categories of observers, aborters, selectors, regulators, confiners and weak intruders form a hierarchy

$$\begin{tikzcd} & \mathcal{A}_a^* \arrow[dr, dash, "\subset"] & & & \\ \mathcal{A}_o^* \arrow[ur, dash, "\subset"] \arrow[dr, dash, "\subset"'] & & \mathcal{A}_r^* \arrow[r, dash, "\subset"] & \mathcal{A}_c^* \arrow[r, dash, "\subset"] & \mathcal{A}_w^* \\ & \mathcal{A}_s^* \arrow[ur, dash, "\subset"'] & & & \end{tikzcd}$$

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.

The examples of aspects discussed in the section 4 remain valid in the non-deterministic case. For instance, debugging aspects are also observer 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 show how to adapt our semantics and properties for non-deterministic programs. We do not (re)present all categories but focus on the two new categories (selectors and regulators) and their corresponding classes of properties.

5.1 Base and woven execution traces

We abstract the base and woven program executions as sets of infinite traces written $\mathcal{B}^*(C_0, \Sigma_0)$ (Definition 8) and $\mathcal{W}^*(C_0, \Sigma_0)$ (Definition 9). As in the deterministic case (*i.e.,* Definitions 1 and 2) we assume programs begin with a dummy $start$ instruction that enables to weave the second (*i.e.,* first after $start$) instruction of the base program. We ignore the $start$ instruction in the traces.

DEFINITION **8.**

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

DEFINITION **9.**

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

5.2 Branching temporal logic CTL*

In the non-deterministic case, classes of properties are subsets of the branching temporal logic CTL* [2]. Grammar 5 defines the positive normal form of CTL* formulae.

GRAMMAR **5.**

$$\begin{aligned}\theta \quad & ::= \quad ap \mid \neg ap \mid \theta_1 \vee \theta_2 \mid \theta_1 \wedge \theta_2 \mid \exists\omega \mid \forall\omega \\ \omega \quad & ::= \quad \theta \mid \omega_1 \vee \omega_2 \mid \omega_1 \wedge \omega_2 \mid \bigcirc\omega \mid \omega_1 \cup \omega_2 \mid \omega_1 W \omega_2\end{aligned}$$

When LTL specifies properties on an execution trace, CTL* specifies properties on a set of execution traces. CTL* extends LTL with 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 5, θ represents properties on trace steps and ω properties on traces.

The semantics of CTL* is quite similar to the semantics of LTL defined in section 3. The semantics of logical quantifiers is defined as follows:

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

In these definitions, the environment T is the set of traces starting from α_j. In our context, T will be initially either $\mathcal{B}^*(C_0, \Sigma_0)$ for α_1 or $\mathcal{W}^*(C_0, \Sigma_0)$ for $\tilde{\alpha}_1$). 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 $\Box$ can be defined in CTL* in the same way as in LTL.

5.3 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 10) 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 **10.**

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

The properties defined by θ^s in Grammar 6 are preserved by selectors.

GRAMMAR **6.**

$$\begin{aligned}\theta^s \quad & ::= \quad sp \mid \neg sp \mid \theta_1^s \vee \theta_2^s \mid \theta_1^s \wedge \theta_2^s \mid \forall\omega^s \\ \omega^s \quad & ::= \quad \theta^s \mid \omega_1^s \vee \omega_2^s \mid \omega_1^s \wedge \omega_2^s \mid \\ & \qquad \omega_1^s \cup \omega_2^s \mid \omega_1^s W \omega_2^s \mid true \cup \omega'^s \\ \omega'^s \quad & ::= \quad ep \mid \neg ep \mid \theta^s \mid \omega_1'^s \vee \omega_2'^s \mid \omega_1'^s \wedge \omega_2'^s \mid \\ & \qquad \omega_1^s \cup \omega_2^s \mid \omega_1^s W \omega_2^s \mid true \cup \omega'^s\end{aligned}$$

Grammar 6 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 5.

THEOREM **5.**

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

Examples of selectors are scheduling aspects or refinement aspects that removes some non-determinism. The scheduling aspects of [8] 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.

5.4 Regulator aspects

Regulators are both aborters and selectors. A regulator (Definition 11) 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 **11.**

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

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 7 are preserved by regulator aspects.

GRAMMAR **7.**

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

Grammar 7 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 6.

THEOREM **6.**

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

6. Related Work

Few people have studied aspect categories and how to reason or ensure properties on aspect-oriented programs.

The starting point of our study is seminal work by Katz [11] 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 [6] 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.* [14] 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 [10] for weakly invasive aspects (weak intruders).

Rinard, Salcianu, and Bugara [17] 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 [3] 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.

7. 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 and classes have been completed and generalized for non-deterministic programs using CTL*.

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.,* [15]) 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.,* [4], [12]) 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) and it would be difficult to ensure that an aspect belongs to that category.

Besides the preservation of properties, our categories have other interesting features. For example, if we assume that observers and aborters only introduce private advice (*i.e.,* that cannot be woven by other aspects) then such aspects cannot interact. They can be composed and woven in any order. The woven program will have the same semantics regardless the order of weaving of such aspects.

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.

Finally, this work does need to be completed by means to determine whether an aspect belongs to a specific category. A possible way is to use static analyses to place aspects within the hierarchy of categories. Katz [11] indicates static checks needed to ensure that an aspect is spectative or regulative. However, checking that an advice does not modify the variables of the base program may involve costly program analyses (*e.g.,* alias analysis). Instead, we are currently working on the design, for each category of aspects, of a domain-specific aspect language ensuring that any aspect written is that language belongs to the corresponding category. The requirements are relatively clear for observers and aborters. The language should ensure the separation of base and aspect states and termination of advice. Aborters are further allowed to use a special `abort` instruction. For confiners and weak intruders, it seems hard to design general domain-specific languages. However, languages to express specific confiners (or weak intruders) such as fault-tolerance or memo aspects can be proposed. All these domain-specific aspect languages, combined with the results of this paper, would guarantee the preservation of key properties by construction.

Acknowledgments

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

References

[1] 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.

[2] 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.

[3] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning, 2002.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

A. Proofs

This appendix presents in some details the proof of Theorem 1. 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).

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

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 1 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 **12.**
$$\begin{array}{l} \forall (C, \Sigma).\ \Sigma^{\psi} \in \mathcal{A}_o \ \Rightarrow\ trace_b(\tilde{\alpha}) = \alpha \\ \textit{with}\ \ \alpha = \mathcal{B}(C, \Sigma^b)\ \ \textit{and}\ \ \tilde{\alpha} = \mathcal{W}(C, \Sigma) \end{array}$$

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 **13.**
$$\begin{array}{l} trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow \\ \forall (j \geq 1).\ \alpha_j = (i_b, \Sigma^b) \Leftrightarrow \tilde{\alpha}_{\tilde{j}} = (i_b, \Sigma) \end{array}$$

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

LEMMA **14.**
$$\begin{array}{l} trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow \\ \quad \forall (i \geq 1).\ \forall (\widetilde{i-1} < j \leq \tilde{i}).trace_b(\tilde{\alpha}_{j\rightarrow}) = \alpha_{i\rightarrow} \end{array}$$

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 1 is shown by proving the more general property

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

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 1 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 1.

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

Base cases

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

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

$$trace_b(\tilde{\alpha}) = \alpha \quad \Rightarrow\ \tilde{\alpha}_{\tilde{1}} = (i_1, \Sigma_1) \quad \text{by Lemma 13}$$

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

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

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

$$\begin{array}{l} \forall (j \geq 1).\ \alpha_{j\rightarrow} \models ep \ \Rightarrow\ \alpha_j \models ep \ \Rightarrow\ m(i_j, ep) \\ \text{By Lemma 13} \\ \alpha_j = (i_j, \Sigma^b) \ \Rightarrow\ \tilde{\alpha}_{\tilde{j}} = (i_j, \Sigma) \\ \text{so}\ \ m(i_j, ep) = m(\tilde{\alpha}_{\tilde{j}}, ep) \\ \text{and}\ \tilde{\alpha}_{\tilde{j}} \models ep \\ \text{and therefore}\ \ \tilde{\alpha}_{\tilde{j}\rightarrow} \models ep \end{array}$$

- $p = \neg sp \in\ \varphi^o$ 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.

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

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

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

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

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

By lemma 14
$trace_b(\tilde{\alpha}) = \alpha \ \Rightarrow \ trace_b(\tilde{\alpha}_{\widetilde{j-1}+1\rightarrow}) = \alpha_{j\rightarrow}$
$\Rightarrow \ \tilde{\alpha}_{\widetilde{j-1}+1\rightarrow} \models \varphi_2^o$ 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 14
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 $\tilde{\alpha} \models \varphi_1^o \cup \varphi_2^o$

∘ $p = \mathit{true} \cup \varphi'^o \ \in \varphi^o$

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

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 $\forall (1 \leq l < \tilde{j}).\ \tilde{\alpha}_{l\rightarrow} \models \mathit{true}$
we have

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

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

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

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

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

∘ $p = \varphi_1^o \cup \varphi_2^o \ \in \varphi'^o$

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

By lemma 14
$trace_b(\tilde{\alpha}) = \alpha \Rightarrow \ trace_b(\tilde{\alpha}_{\widetilde{k-1}+1\rightarrow}) = \alpha_{k\rightarrow}$
$\Rightarrow \ \tilde{\alpha}_{\widetilde{k-1}+1\rightarrow} \models \varphi_2^o$ 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 14
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

$$\begin{array}{ll} \text{Thus} & \exists (m \geq \tilde{j} \geq j).\tilde{\alpha}_{m\rightarrow} \models \varphi_2^o \\ \wedge & \forall (j \leq \tilde{j} \leq n < m).\tilde{\alpha}_{n\rightarrow} \models \varphi_1^o \end{array}$$

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

∘ $p = \mathit{true} \cup \varphi'^o \ \in \varphi'^o$

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

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

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

then

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

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

□