Interference of Larissa Aspects

David Stauch
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
David.Stauch@imag.fr

Karine Altisen
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
Karine.Altisen@imag.fr

Florence Maraninchi
Verimag/INPG, Grenoble
Centre Equation – 2, avenue
de Vignate, F 38610 GIERES
Florence.Maraninchi@imag.fr

ABSTRACT

Aspect Oriented Programming is a programming language concept for expressing cross-cutting concerns. A key point when dealing with aspects is the notion of interference. Applying several aspects to the same program may lead to unintended results because of conflicts between the aspects. In this paper, we study the notion of interference for Larissa, a formally defined language. Larissa is the aspect extension of Argos, a StateChart-like automata language designed to program reactive systems. We present a way to weave several aspects in a less conflict-prone manner, and a means to detect remaining conflicts statically, at a low complexity.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Theory, Verification

Keywords

reactive systems, aspect-oriented programming, formal semantics, synchronous languages, aspect interference

1. INTRODUCTION

Aspect oriented programming (AOP). AOP has emerged recently. It aims at providing new facilities to implement or modify existing programs: it may be the case that implementing some new functionality or property in a program P can not be done by adding a new module to the existing structure of P but rather by modifying every module in P. This kind of functionality or property is then called an *aspect*. AOP provides a way to define aspects separately from the rest of the program and then to introduce or "weave" them automatically into the existing structure. Many programming languages have been, now, extended with aspects.

Larissa. In this paper, we study the notion of aspect interferences for a formally defined language called Larissa. Larissa [1] is an aspect extension for Argos [13], a language used to program reactive systems. Argos pertains to the synchronous languages family; it is based on Mealy machines that communicate via Boolean signals, plus a simple set of atomic operators on the machines. The semantics of Argos programs is formally defined. Larissa defines aspects that are woven into Argos programs. The selection of join points is based on temporal pointcuts and the advice is some modification of the transitions of the basic machines. The weaving of Larissa aspects preserves the equivalence between programs.

Interferences. A key point when dealing with aspects is the notion of interferences. If A_1 and A_2 are aspects, and weaving first A_1 and then A_2 yields a different program than weaving first A_2 and then A_1, A_1 and A_2 are said to interfere. Aspect interference may depend on how the weaver proceeds: if it sequentially weaves first A_1 into a program P and then A_2 into the result — denoted by $P \triangleleft A_1 \triangleleft A_2$ — or if it weaves A_1 and A_2 together into P — denoted by $P \triangleleft \{A_1, A_2\}$.

In general, sequential weaving often causes interference. This may be one reason why languages such as AspectJ do not proceed sequentially. As an explanation, let us look at the following example. The class `Test` has a method `foo` and a method `main`, which calls `foo` on some `Test` object.

```
class Test {
  public void foo () { ... }
  public static void main (String [] args)
    { (new Test ()).foo (); }              }
```

We then define the aspect `A1` that has a method `bar()` and adds a call to `bar` at the end of every call to every method named `foo`.

```
aspect A1 {
    void bar () { ... }
    after(): call(* foo (..)) { bar ();}   }
```

After compiling together the class `Test` and the aspect `A1` (`Test`$\triangleleft$ `A1`), the execution of `main` executes `foo ()` and then `bar ()`. Let us introduce another aspect `A2` which adds some code at the end of every method named `bar`.

```
aspect A2 { after(): call(* bar (..)) { XXX } }
```

If the class `Test` is compiled with `A2` only (`Test`$\triangleleft$ `A2`), nothing changes (the class `Test` is unchanged, since no method `bar` exists, we have `Test` = `Test`$\triangleleft$ `A2`).

Now imagine that a weaver for AspectJ produces Java code as a backend, and that for weaving two aspects, we first weave the first one, obtain some Java code, and weave the second aspect into the result. We call this sequential weaving of aspects. Larissa works this way: as the other Argos operators, aspect weaving is defined as the transformation of an Argos program into another Argos program.

Sequentially weaving `A1` into `Test` and then `A2` into the result provides a different program from weaving first `A2` and then `A1`. If we execute the `main` method in both cases, (`Test`$\triangleleft$ `A1`)$\triangleleft$ `A2` executes `foo`, `bar` and then the code `XXX` added by `A2`, whereas (`Test`$\triangleleft$ `A2`)$\triangleleft$ `A1` only executes `foo` and `bar`. `A2` is activated in the first case, but not in the second.

AspectJ does not work this way. Join points are defined as points in the execution of the woven program, including those contained in advice. Thus, aspects affect each other, and cannot be woven sequentially. They must be woven together, i.e., for the example, `Test`$\triangleleft$ {`A1`, `A2`}. In the example, this produces the same result as (`Test`$\triangleleft$ `A1`)$\triangleleft$ `A2`.

In this paper we propose a weaving mechanism for Larissa that weaves several aspects together into a program, and thus eliminates some cases of interference. As opposed to AspectJ, pointcuts do not capture join points in the woven program, but in the base program. In Larissa, advice only affects the base program, whereas in AspectJ, advice also affects advice.

Aspects in AspectJ may still interfere. This is illustrated by the second example:

```
aspect A3 {
  declare precedence : A4, A3;     // (**)
  before(): call(* foo (..)) { ... }  }
aspect A4 {
  before(): call(* foo (..)) { ... }  }
```

The sets of join points selected by `A3` and by `A4` are the same. The interference here is unavoidable since the advice programs have to be executed sequentially. In such a case, AspectJ allows to describe the order of application of the advice (see line `(**)`).

Likewise, aspects in Larissa may still interfere if they share join points, even if they are woven together into a program. We further analyze interference for aspects that are woven together. We present sufficient conditions to prove non-interference, either for two aspects in general or two aspects and a specific program.

Section 2 presents the Argos language and the Larissa extension, Section 3 illustrates the language on an example, Section 4 deals with interferences for Larissa, Section 5 explores some related work and Section 6 gives some conclusions and perspectives.

2. LANGUAGE

This section presents a restriction of the Argos language [13], and the Larissa extension [1]. The Argos language is defined as a set of operators on complete and deterministic input/output automata communicating via Boolean signals. The semantics of an Argos program is given as a trace semantics that is common to a wide variety of reactive languages.

2.1 Traces and trace semantics

Definition 1 (Traces) *Let $\mathcal{I}$, $\mathcal{O}$ be sets of Boolean input and output variables representing signals from and to the environment. An* input trace*, it, is a function: it : $\mathbb{N} \longrightarrow [\mathcal{I} \longrightarrow \{\mathit{true}, \mathit{false}\}]$. An* output trace*, ot, is a function: ot : $\mathbb{N} \longrightarrow [\mathcal{O} \longrightarrow \{\mathit{true}, \mathit{false}\}]$. We denote by $InputTraces$ (resp. $OutputTraces$) the set of all input (resp. output) traces. A pair (it, ot) of input and output traces (i/o-traces for short) provides the valuations of every input and output at each instant $n \in \mathbb{N}$. We denote by $it(n)[i]$ (resp. $ot(n)[o]$) the value of the input $i \in \mathcal{I}$ (resp. the output $o \in \mathcal{O}$) at the instant $n \in \mathbb{N}$.*

A set of pairs of i/o-traces $S = \{(it, ot) \mid it \in InputTraces \wedge ot \in OutputTraces\}$ is deterministic *iff $\forall (it, ot), (it', ot') \in S \,.\, (it = it') \implies (ot = ot')$.*

A set of pairs of i/o traces $S = \{(it, ot) \mid it \in InputTraces \wedge ot \in OutputTraces\}$ is complete *iff $\forall it \in InputTraces \,.\, \exists ot \in OutputTraces \,.\, (it, ot) \in S$.*

A set of traces is a way to define the semantics of an Argos program P, given its inputs and outputs. From the above definitions, a program P is *deterministic* if from the same sequence of inputs it always computes the same sequence of outputs. It is *complete* whenever it allows every sequence of every eligible valuations of inputs to be computed. Determinism is related to the fact that the program is indeed written with a programming language (which has deterministic execution); completeness is an intrinsic property of the program that has to react forever, to every possible inputs without any blocking.

2.2 Argos

The core of Argos is made of input/output automata, the synchronous product, and the encapsulation.

The synchronous product allows to put automata in parallel which synchronize on their common inputs. The encapsulation is the operator that expresses the communication between automata with the synchronous broadcast: if two automata are put in parallel, they can communicate via some signal s. This signal is an input of the first automaton and an output of the second. The encapsulation operator computes this communication and then hides the signal s.

The semantics of an automaton is defined by a set of traces, and the semantics of the operators is given by translating expressions into flat automata.

Definition 2 (Automaton) *An* automaton $\mathcal{A}$ *is a tuple $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{Q}$ is the set of states, $s_{init} \in \mathcal{Q}$ is the initial state, $\mathcal{I}$ and $\mathcal{O}$ are the sets of Boolean input and output variables respectively, $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $\mathcal{B}ool(\mathcal{I})$ denotes the set of Boolean formulas with variables in $\mathcal{I}$. For $t = (s, \ell, O, s') \in \mathcal{T}$, $s, s' \in \mathcal{Q}$ are the source and target states, $\ell \in \mathcal{B}ool(\mathcal{I})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.*

Complete monomials are conjunctions that for each $i \in \mathcal{I}$ contain either i or $\bar{i}$. Requiring complete monomials as input labels makes the definition of the operators easier. A transition with an arbitrary input label can be easily converted in a set of transitions with complete monomials, and

can thus be considered as a macro notation. We will use such transitions in the examples.

The *semantics* of an automaton $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is given in terms of a set of pairs of i/o-traces. This set is built using the following functions:

$$S\_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times \mathbb{N} \longrightarrow \mathcal{Q}$$
$$O\_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times \mathbb{N} \setminus \{0\} \longrightarrow 2^{\mathcal{O}}$$

$S\_step(s, it, n)$ is the state reached from state s after performing n steps with the input trace it; $O\_step(s, it, n)$ are the outputs emitted at step n:

$$n = 0 : \; S\_step_{\mathcal{A}}(s, it, n) = s$$
$$n > 0 : \; S\_step_{\mathcal{A}}(s, it, n) = s' \qquad O\_step_{\mathcal{A}}(s, it, n) = O$$
$$\text{where } \exists (S\_step_{\mathcal{A}}(s, it, n-1), \ell, O, s') \in \mathcal{T}$$
$$\wedge\ \ell \text{ has value true for } it(n-1)\ .$$

We note $Traces(\mathcal{A})$ the set of all traces built following this scheme: $Traces(\mathcal{A})$ defines the semantics of $\mathcal{A}$. The automaton $\mathcal{A}$ is said to be *deterministic* (resp. *complete*) iff its set of traces $Traces(\mathcal{A})$ is deterministic (resp. complete) (see Definition 1). Two automata $\mathcal{A}_1$, $\mathcal{A}_2$ are *trace-equivalent*, noted $\mathcal{A}_1 \sim \mathcal{A}_2$, iff $Traces(\mathcal{A}_1) = Traces(\mathcal{A}_2)$.

Definition 3 (Synchronous Product) *Let* $\mathcal{A}_1 = (\mathcal{Q}_1, s_{init1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ *and* $\mathcal{A}_2 = (\mathcal{Q}_2, s_{init2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ *be automata. The* synchronous product *of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *is the automaton* $\mathcal{A}_1 \| \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{init1} s_{init2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ *where* $\mathcal{T}$ *is defined by:*

$$(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \Longleftrightarrow$$
$$(s_1 s_2, \ell_1 \wedge \ell_2, O_1 \cup O_2, s_1' s_2') \in \mathcal{T}\ .$$

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and completeness.

Definition 4 (Encapsulation) *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ *be a set of inputs and outputs of* $\mathcal{A}$. *The* encapsulation *of* $\mathcal{A}$ *w.r.t.* Γ *is the automaton* $\mathcal{A} \setminus \Gamma = (\mathcal{Q}, s_{init}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ *where* $\mathcal{T}'$ *is defined by:*

$$(s, \ell, O, s') \in \mathcal{T} \wedge \ell^+ \cap \Gamma \subseteq O \wedge \ell^- \cap \Gamma \cap O = \emptyset \Longleftrightarrow$$
$$(s, \exists \Gamma\ .\ \ell, O \setminus \Gamma, s') \in \mathcal{T}'$$

ℓ^+ *is the set of variables that appear as positive elements in the monomial* ℓ *(i.e.* $\ell^+ = \{x \in \mathcal{I} \mid (x \wedge \ell) = \ell\}$*).* ℓ^- *is the set of variables that appear as negative elements in the monomial* l *(i.e.* $\ell^- = \{x \in \mathcal{I} \mid (\neg x \wedge \ell) = \ell\}$*).*

Intuitively, a transition $(s, \ell, O, s') \in \mathcal{T}$ is still present in the result of the encapsulation operation if its label satisfies a local criterion made of two parts: $\ell^+ \cap \Gamma \subseteq O$ means that a local variable which needs to be true has to be emitted by the same transition; $\ell^- \cap \Gamma \cap O = \emptyset$ means that a local variable that needs to be false should *not* be emitted in the transition.

If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by $\exists \Gamma\ .\ \ell$ for the input part, and by $O \setminus \Gamma$ for the output part.

In general, the encapsulation operation does not preserve determinism nor completeness. This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [4]).

An example

Figure 1 (a) shows a 3-bits counter. Dashed lines denote parallel compositions and the overall box denotes the encapsulation of the three parallel components, hiding signals `b` and `c`. The idea is the following: the first component on the right receives `a` from the environment, and sends `b` to the second one, every two `a`'s. Similarly, the second one sends `c` to the third one, every two `b`'s. `b` and `c` are the carry signals. The global system has `a` as input and `d` as output; it counts `a`'s modulo 8, and emits `d` every 8 `a`'s. Applying the semantics of the operator (first the product of the three automata, then the encapsulation) yields the simple flat automaton with 8 states (Figure 1 (b)).

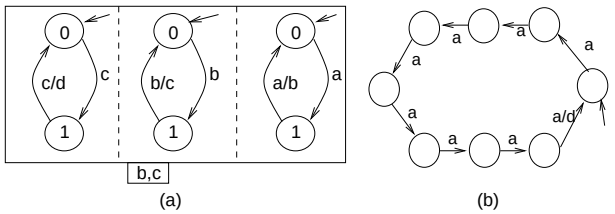

Figure 1: A 3-bits counter. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering cond. / outputs emitted`", e.g. the transition labelled by "`a/b`" is triggered when `a` is true and emits `b`.

2.3 Larissa

Argos operators are already powerful. However, there are cases in which they are not sufficient to modularize all concerns of a program: some small modifications of the global program's behavior may require that we modify all parallel components, in a way that is not expressible with the existing operators.

Therefore, we proposed Larissa [1], an aspect-oriented extension for Argos, which allows the modularization of a number of recurrent problems in reactive programs. As most aspect languages, Larissa contains a pointcut and an advice construct. The pointcut selects a number of transitions, called *join point transitions*, from an automaton, and the advice modifies these transitions, both their outputs and their target state.

In this paper, we only present a simple kind of advice, in which the target state is the same for all join point transitions. In [1, 2] we present more sophisticated kinds of advice, which allow to jump forward or backward from the join point transition, and to replace join point transitions with complete automata. We believe that the ideas presented in this paper can be extended easily to the other kinds of advice, as the join point mechanism is the same.

We ensure the semantic properties that make it possible to introduce aspects as a normal operator into Argos. Specifically, as shown in [1], determinism and completeness are preserved, as well as semantic equivalence between programs: when we apply the same aspect to two trace-equivalent programs, we obtain two trace-equivalent programs.

Specifying pointcuts. Because we want to preserve trace equivalence, we cannot express pointcuts in terms of the internal structure of the base program. For instance, we do not allow pointcuts to refer explicitly to state names (as AspectJ [9] can refer to the name of a private method). As a consequence, pointcuts may refer to the observable behavior of the program only, i.e., its inputs and outputs. In the family of synchronous languages, where the communication between parallel components is the synchronous broadcast, *observers* [8] are a powerful and well-understood mechanism which may be used to describe pointcuts. An observer is a program that may observe the inputs and the outputs of the base program, without modifying its behavior, and compute some safety property (in the sense of safety/liveness properties as defined in [11]).

We use an observer P_{JP} that emits a special output `JP` to describe a pointcut. Whenever P_{JP} emits `JP`, "we are in" a join point, and the woven program executes the advice.

Join Point Weaving. If we simply put a program P and an observer P_{JP} in parallel, P's outputs $\mathcal{O}$ will become synchronization signals between them, as they are also inputs of P_{JP}. They will be encapsulated, and are thus no longer emitted by the product. We avoid this problem by introducing a new output o' for each output o of P: o' will be used for the synchronization with P_{JP}, and o will still be visible as an output. First, we transform P into P' and P_{JP} into P'_{JP}, where $\forall o \in \mathcal{O}$, o is replaced by o'. Second, we duplicate each output of P by putting P in parallel with one single-state automaton per output o defined by: $dupl_o = (\{q\}, q, \{o'\}, \{o\}, \{(q, o', o, q)\})$. The complete product, where $\mathcal{O}$ is noted $\{o_1, ..., o_n\}$, is given by:

$$\boldsymbol{\mathcal{P}}(P, P_{JP}) = (P' \| P'_{JP} \| dupl_{o_1} \| \ ... \ \| dupl_{o_n}) \setminus \{o'_1, ..., o'_n\}$$

The program $\boldsymbol{\mathcal{P}}(P, P_{JP})$ is first transformed into the single trace-equivalent automaton by applying the definition of the operators. We use the same notation, $\boldsymbol{\mathcal{P}}(P, P_{JP})$, for the program and its transformation. Then, the join points are selected: they are the *transitions* of $\boldsymbol{\mathcal{P}}(P, P_{JP})$ that emit *JP*.

Specifying the advice. A piece of advice modifies the join point transitions: it redefines their target states and their outputs. The only advice presented in this paper specifies the target state of the join point transition globally, by a finite input trace σ. When executing σ on $\boldsymbol{\mathcal{P}}(P, P_{JP})$ from its initial state, this leads to some state of $\boldsymbol{\mathcal{P}}(P, P_{JP})$, *targ*. *targ* is the unique new target state for any join point transition.

Advice Weaving. Advice weaving consists in changing the target state of the join point transitions to the single state specified by the finite input trace σ, and in replacing their outputs by the *advice outputs* O_{adv}.

Definition 5 (Advice weaving) *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $adv = (O_{adv}, \sigma)$ *a piece of advice, with* $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *a finite input trace of length* $\ell_\sigma + 1$. *The advice weaving operator,* $\triangleleft_{JP}$, *weaves asp on* $\mathcal{A}$ *and returns the automaton* $\mathcal{A} \triangleleft_{JP} adv = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O} \cup O_{adv}, \mathcal{T}')$, *where* $\mathcal{T}'$ *is defined as follows, with* $targ = S\_step_{\mathcal{A}}(s_{init}, \sigma, \ell_\sigma)$ *being the new target state:*

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \quad (1)$$

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\big) \implies (s, \ell, O_{adv}, targ) \in \mathcal{T}' \quad (2)$$

Transitions (1) are not join point transitions and are left unchanged. Transitions (2) are the join point transitions, their final state *targ* is specified by the finite input trace σ. $S\_step_{\mathcal{A}}$ (which has been naturally extended to finite input traces) executes the trace during ℓ_σ steps, from the initial state of $\mathcal{A}$.

General aspect definition. Putting together the pointcut and the advice, we define an aspect as follows:

Definition 6 (Larissa aspect) *An aspect, for a program* P *on inputs* $\mathcal{I}$ *and outputs* $\mathcal{O}$, *is a tuple* (P_{JP}, adv) *where*

- $P_{JP} = (\mathcal{Q}_{pc}, s_{init}, \mathcal{I} \cup \mathcal{O}, \{JP\} \cup O_{pc}, \mathcal{T}_{pc})$ *is the pointcut program, and JP occurs nowhere else in the environment.*
- $adv = (O_{adv}, \sigma)$ *is the advice, which contains two items:*
  - O_{adv} *is the set of outputs emitted by the advice transitions, which may contain fresh variables as well as elements of* $\mathcal{O}$.
  - $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *is a finite input trace of length* $\ell_\sigma + 1$. *It defines the single target state of the advice transitions by executing the trace from the initial state.*

An aspect is woven into a program by first determining the join point transitions and then weaving the advice.

Definition 7 (Aspect weaving) *Let* P *be a program and* $asp = (P_{JP}, adv)$ *an aspect for* P. *The weaving of asp on* P *is defined as follows:*

$$P \triangleleft asp = \boldsymbol{\mathcal{P}}(P, P_{JP}) \triangleleft_{JP} adv \, .$$

3. EXAMPLE

As an example, we present a simplified view of the interface of a complex wristwatch, implemented with Argos and Larissa. The full case study was presented in [2]. The interface is a modified version of the Altimax[1] model by Suunto[1].

3.1 The Watch

The Altimax wristwatch has an integrated altimeter, a barometer and four buttons, the `mode`, the `select`, the `plus`, and the `minus` button. Each of the main functionalities (time keeping, altimeter, barometer) has an associated main mode, which displays information, and a number of submodes, where the user can access additional functionalities. An Argos program that implements the interface of the watch is shown in Figure 2. For better readability, only those state names, outputs and transitions we will refer to are shown.

In a more detailed model (as in [2]) the submode states would contain behavior using the refinement operator of Argos (see [13] for a definition). We choose not to present this

[1]Suunto and Altimax are trademarks of Suunto Oy.

operator in this paper since we do not need it to define aspect weaving. Adding refinement changes nothing for the weaving definition, as it works directly on the transformation of the program into a single trace-equivalent automaton. For the same reason, the interference analysis presented in Section 4 is also the same.

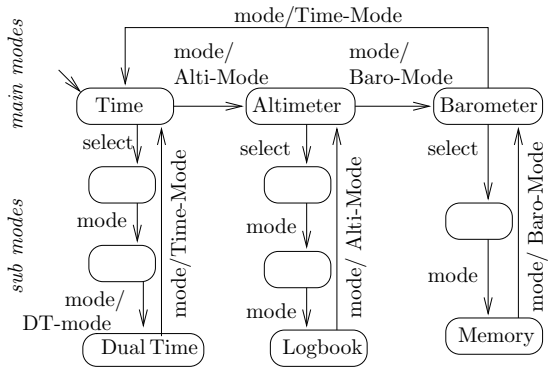

Figure 2: The Argos program for the Altimax watch.

The buttons of the watch are the inputs of the program. The `mode` button circles between modes, the `select` button selects the submodes. There are two more buttons: the `plus` and the `minus` button which modify current values in the submodes, but their effect is not shown in the figure. The buttons have different meanings depending on the mode in which the watch is currently.

The interface component we model here interprets the meaning of the buttons the user presses, and then calls a corresponding function in an underlying component. The outputs are commands to that component. E.g., whenever the program enters the Time Mode, it emits the output `Time-Mode`, and the underlying component shows the time on the display of the watch.

3.2 Two Shortcut Aspects

The `plus` and the `minus` buttons have no function consistent with their intended meaning in the main modes: there are no values to increase or decrease. Therefore, they are given a different function in the main modes: when one presses the `plus` or the `minus` button in a main mode, the watch goes to a certain submode. The role of the `plus` and `minus` buttons in the main modes are called *shortcuts* since it allows to quickly activate a functionnality, which would have needed, otherwise, a long sequence of buttons.

Pressing the `plus` button in a main mode activates the `logbook` function of the altimeter, and pressing the `minus` button activates the 4-day `memory` of the barometer. These functions are quite long to reach without the shortcuts since the logbook is the third submode of the altimeter, and the 4-day memory is the second submode of the barometer.

These shortcuts can be implemented easily with Larissa aspects. Figure 3 (a) shows the pointcut for the logbook aspect, and Figure 3 (b) the pointcut for the memory aspect. In both pointcuts, state `main` represents the main modes and state `sub` represents the submodes. When, in a main mode, `plus` (resp. `minus`) is pressed, the pointcut emits JP_l (resp. JP_m), thus the corresponding advice is executed; when `select` is pressed, the pointcut goes to the `sub` state, so as to record that the shortcuts are no longer active and

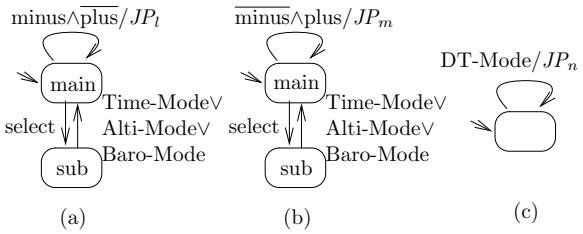

Figure 3: The pointcuts for the aspects.

that the `plus` and `minus` buttons have their usual meaning. As advice, we specify the trace that leads to the functionality we want to reach, i.e. σ_l =`mode.select.mode.mode` for the logbook aspect and σ_m =`mode.mode.select.mode` for the 4-day memory aspect, and the output that tells the underlying component to display the corresponding information.

3.3 The No-DTM Aspect

We want to reuse the interface program of the Altimax to build the interface of another wristwatch, which differs from the Altimax in that it has no Dual-Time mode (third submode of the main mode Time). Therefore, we write an aspect that removes the Dual-Time mode from the interface: all incoming transitions are redirected to another state. Figure 3 (c) shows the pointcut for the No-DTM aspect. It selects all transitions that emit `DT-Mode`, the output that tells the underlying component to show the information corresponding to the Dual-Time mode on the display. Because the Dual-Time mode is the last submode of the Time mode, we want the join point transitions to point to the Time main mode, i.e. the initial state of the program. Thus, as advice, we specify `Time-Mode` as output and an empty trace, which points to the Time main mode.

4. INTERFERENCE

This section identifies problems that occur when several aspects are applied to a program, and, as a solution, proposes to weave several aspects at the same time. A mechanism to prove that no interferences remain is also proposed.

4.1 Applying Several Aspects

If we apply first the logbook aspect and then, sequentially, the memory aspect to the watch program, the aspects do not behave as we would expect. If, in the woven program, we first press the `minus` button in a main mode, thus activating the logbook aspect, and then the `plus` button, the memory aspect is activated, although we are in a sub mode. This behavior was clearly not intended by the programmer of the memory aspect.

The problem is that the memory aspect has been written for the program without the logbook aspect: the pointcut assumes that the only way to leave a main mode is to press the `select` button. However, the logbook aspect invalidates that assumption by adding transitions from the main modes to a submode. When these transitions are taken, the pointcut of the memory aspect incorrectly assumes that the program is still in a main mode.

Furthermore, for the same reason, applying first the memory aspect and then the logbook aspect produces (in terms of trace-equivalence) a different program from applying first the logbook aspect and then the memory aspect:

`watch` $\triangleleft$ `logbook` $\triangleleft$ `memory` $\nsim$ `watch` $\triangleleft$ `memory` $\triangleleft$ `logbook`.

As a first attempt to define *aspect interference*, we say that two aspects $\mathcal{A}_1$ and $\mathcal{A}_2$ interfere when their application on a program P in different orders does not yield two trace-equivalent programs: $P \triangleleft \mathcal{A}_1 \triangleleft \mathcal{A}_2 \nsim P \triangleleft \mathcal{A}_2 \triangleleft \mathcal{A}_1$. We say that two aspects that do not interfere are *independent*.

With interfering aspects, the aspect that is woven second must know about the aspect that was applied first. To be able to write aspects as the ones above independently from each other, we propose a mechanism to weave several aspects at the same time. The idea is to first determine the join point transitions for all the aspects, and then apply the advice.

Definition 8 (Joint weaving of several aspects) *Let $\mathcal{A}_1 \ldots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$, and P a program. We define the application of $\mathcal{A}_1 \ldots \mathcal{A}_n$ on P as follows:*

$$P \triangleleft (\mathcal{A}_1, \ldots, \mathcal{A}_n) = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \triangleleft_{JP_n} adv_n \ldots \triangleleft_{JP_1} adv_1$$

Jointly weaving the logbook and the memory aspect leads to the intended behavior, and the weaving order does not influence the result, because both aspects first select their join point transitions in the main modes, and change the target states of the join point transitions only afterwards.

Note that Definition 8 does not make sequential weaving redundant. We still need to weave aspects sequentially in some cases, when the second aspects must be applied to the result of the first. For instance, imagine an aspect that adds an additional main mode to the watch (with a kind of advice not presented in this paper). Then, the shortcut aspects must be sequentially woven *after* this aspect, so that they can select the new main mode as join point.

Definition 8 does not solve all conflicts. Indeed, the $\mathcal{A}_i$ in $P \triangleleft (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ do not commute, in general, since the advice weaving is applied sequentially. We define aspect interference for the application of several aspects.

Definition 9 (Aspect Interference) *Let $\mathcal{A}_1$ ... $\mathcal{A}_n$ be some aspects, and P a program. We say that $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$* interfere *for P iff*

$$P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \nsim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n)$$

As an example for interfering aspects, assume that the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (a)) is only `minus` and the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (b)) is only `plus`. In this case, the two aspects share some join point transitions, namely when both buttons are pressed at the same time in a main mode. Both aspects then want to execute their advice, but only one can, thus they interfere. Only the aspect that was applied last is executed.

In such a case, the conflict should be made explicit to the programmer, so that it can be solved by hand. Here, it was resolved by changing the pointcuts to the form they have in Figure 3, so that neither aspect executes when both buttons are pressed.

4.2 Proving Non-Interference

In this section, we show that in some cases, non-interference of aspects can be proven, if the aspects are woven jointly, as defined in Definition 8. We can prove non-interference of two given aspects either for any program, or for a given program. Following [6], we speak of *strong independence* in the first case, and of *weak independence* in the second.

We use the operator *advTrans* to determine interference between aspects. It computes all the join point transitions of an automaton, i.e. all transitions with a given output *JP*.

Definition 10 *Let $A = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $JP \in \mathcal{O}$. Then,*

$$advTrans(A, JP) = \{t | t = (s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\} .$$

The following theorem proves strong independence between two aspects.

Theorem 1 (Strong Independence) *Let $\mathcal{A}_1 \ldots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$. Then, the following equation holds:*

$$\begin{aligned} &advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_i) \\ &\cap\, advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_{i+1}) = \emptyset \\ &\Rightarrow P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n) \end{aligned}$$

See appendix A for a proof. Theorem 1 states that if there is no transition with both JP_i and JP_{i+1} as outputs in the product of P_{JP_i} and $P_{JP_{i+1}}$, $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ are independent and thus can commute while weaving their advice. Theorem 1 defines a sufficient condition for non-interference, by looking only at the pointcuts. When the condition holds, the aspects are said to be *strongly independent*.

Theorem 2 (Weak Independence) *Let $\mathcal{A}_1 \ldots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$, and $P_{pc} = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$. Then, the following equation holds:*

$$\begin{aligned} &advTrans(P_{pc}, JP_i) \cap advTrans(P_{pc}, JP_{i+1}) = \emptyset \\ &\Rightarrow P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n) \end{aligned}$$

See appendix B for a proof. Theorem 2 states that if there is no transition with both JP_i and JP_{i+1} as outputs in P_{pc}, $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$ do not interfere. This is weaker than Theorem 1 since it also takes the program P into account. However, there are cases in which the condition of Theorem 1 is false (thus it yields no results), but Theorem 2 allows to prove non-interference. See Section 4.4 for an example.

Theorem 2 is a sufficient condition, but, as Theorem 1, it is not necessary: it may not be able to prove independence for two independent aspects. The reason is that it does not take into account the effect of the advice weaving: consider two aspects such that the only reason why the condition for Theorem 2 is false is a transition sourced in some state s, and such that s is only reachable through another join point transition; if the advice weaving makes this state unreachable, then the aspects do not interfere.

The results obtained by both Theorems are quite intuitive. They mean that if the pointcut mechanism does not select any join points common to two aspects, then these aspects do not interfere. This condition can be calculated on the pointcuts alone, or can also take the program into account.

Note that the detection of non-interference is a static condition that does not add any complexity overhead. Indeed, to weave the aspects, the compiler needs to build first

$P_{JP_1} \| \ldots \| P_{JP_n} = P_{\text{all } JP}$: the condition of Theorem 1 can be checked during the construction of $P_{\text{all } JP}$. Second, the weaver builds $P_{\text{pc}} = \boldsymbol{\mathcal{P}}(P, P_{\text{all } JP})$. Afterwards, it can check the condition of Theorem 2. Thus, to calculate the conditions of both Theorems, it is sufficient to check the outputs of the transitions of intermediate products during the weaving. The weaver can easily emit a warning when a potential conflict is detected.

To have an exact characterization of non-interference, it is still possible to compute the predicate $P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_i, \mathcal{A}_{i+1} \ldots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \ldots \mathcal{A}_{i+1}, \mathcal{A}_i \ldots \mathcal{A}_n)$, but calculating semantic equality is very expensive for large programs.

Note that the interference presented here only applies to the joint weaving of several aspects, as defined in Definition 8. Sequentially woven aspects may interfere even if their join points are disjoint, because the pointcut of the second aspects applies to the woven program. A similar analysis to prove non-interference of sequential weaving would be more difficult, because the effect of the advice must be taken into account. Moreover, it is not clear when such an analysis makes sense: sequential weaving should be used only if one aspect depends on the other, and interference is unavoidable.

4.3 Interference between the Shortcut Aspects

Figure 4 (a) shows the product of the pointcuts of the logbook and the memory aspect. There are no transitions that emit both JP_l and JP_m, thus, by applying Theorem 1, we know that the aspects do not interfere, independently of the program they are applied to.

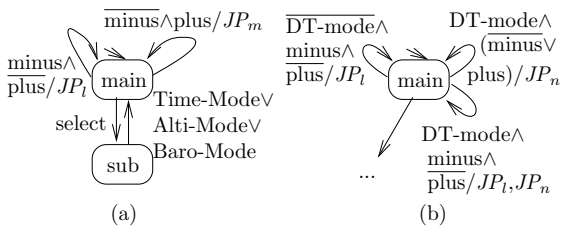

Figure 4: Interference between pointcuts.

Let us assume again that the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (a)) is only `minus` and the condition of the join point transition of the pointcut of the logbook aspect (Figure 3 (b)) is only `plus`. In this case, the state main in Figure 4 (a) would have another loop transition, with label `minus∧plus/`JP_l`,`JP_m. Thus, Theorem 1 not only states that the aspects potentially interfere, but it also gives a means to determine where: here, the problem is that when both `minus` and `plus` are pressed in a main mode, at the same time, both aspects are activated. Larissa thus emits a warning and the user is invited to solve the conflict if needed.

4.4 Interference between a Shortcut and the No-DTM Aspect

Figure 4 (b) shows the initial state of the product of the pointcuts of the logbook (Figure 3 (a)) and the No-DTM aspect (Figure 3 (c)). There is a transition that has both JP_l and JP_n as outputs. Theorem 1 states that the aspects may interfere, but when applied to the wristwatch controller, they do not. This is because the `DT-mode` is an output of the controller and is never emitted when the watch is in a main mode, where the logbook aspect can be activated. As the `DT-mode` is always false in the main modes, the conflicting transition is never enabled. When applied to another program, however, the aspects may interfere.

In this example, the use of Theorem 2 is thus needed to show that the aspects do not interfere when applied to the wristwatch controller. Its condition is true, as expected, because JP_l is only emitted in the main modes, and JP_n only in the Time submodes.

5. RELATED WORK

Some authors discuss the advantages of sequential vs. joint weaving. Lopez-Herrejon and Batory [12] propose to use sequential weaving for incremental software development. Colyer and Clement [5, Section 5.1] want to apply aspects to bytecode which already contains woven aspects. In AspectJ, this is impossible because the semantics would not be the same as weaving all aspects at the same time.

Sihman and Katz [15] propose SuperJ, a superimposition language which is implemented through a preprocessor for AspectJ. They propose to combine superimpositions into a new superimposition, either by sequentially applying one to the other, or by combining them without mutual influence. Superimpositions contain assume/guarantee contracts, which can be used to check if a combination is valid.

A number of authors investigate aspect interference in different formal frameworks. Much of the work is devoted to determining the correct application order for interfering aspects, whereas we focus on proving non-interference.

Douence, Fradet, and Südholt [6] present a mechanism to statically detect conflicts between aspects that are applied in parallel. Their analysis detects all join points where two aspects want to insert advice. To reduce the detection of spurious conflicts, they extend their pointcuts with shared variables, and add constraints that an aspect can impose on a program. To resolve remaining conflicts, the programmer can then write powerful composition adaptors to define how the aspects react in presence of each other.

Pawlak, Duchien, and Seinturier [14] present a way to formally validate precedence orderings between aspects that share join points. They introduce a small language, CompAr, in which the user expresses the effect of the advice that is important for aspect interaction, and properties that should be true after the execution of the advice. The CompAr compiler can then check that a given advice ordering does not invalidate a property of an advice.

Durr, Staijen, Bergmans, and Aksit [7] propose an interaction analysis for Composition Filters. They detect when one aspect prevents the execution of another, and can check that a specified trace property is ensured by an aspect.

Balzarotti, Castaldo D'Ursi, Cavallaro and Monga [3] use program slicing to check if different aspects modify the same code, which might indicate interference.

Clean interfaces which take aspects into account can also help to detect interferences. E.g., aspect-aware interfaces [10] indicate where two aspects advise the same methods in a system.

6. CONCLUSION

We present an analysis for aspect interference for a simple

but significant part of Larissa. We expect that it can be applied without major modifications to the rest of Larissa. First, we introduced an additional operator which jointly weaves several aspects together into a program, closer to the way AspectJ weaves aspects. Because Larissa is defined modularly, we only had to rearrange the building steps of the weaving process. Then, we could analyze interference with a simple parallel product of the pointcuts.

When a potential conflict is detected, the user has to solve it by hand, if needed. In the examples we already studied, the conflicts were solved by simple modifications of the pointcut programs. We plan to further explore this problem, but we believe no new language construct will be needed.

It seems that the interference analysis for Larissa is quite precise, i.e. we can prove independence for most independent aspects. One reason for that are Larissa's powerful pointcuts, which describe join points statically, yet very precisely, on the level of transitions. Another reason is the exclusive nature of the advice. Two pieces of advice that share a join point transition never execute sequentially, but there is always one that is executed while the other is not. If the two pieces of advice are not equivalent, this leads to a conflict. Thus, as opposed to [6], assuming that a shared join point leads to a conflict does not introduce spurious conflicts.

In addition, because our language is connected to formal verification tools, we can check whether different aspect orderings result in trace-equivalent automata. This, however, is only possible because Argos is restricted to Boolean signals; otherwise trace-equivalence is not decidable. It would be interesting to design an approximate interference analysis for Larissa aspects in the presence of valued signals.

7. REFERENCES

[1] K. Altisen, F. Maraninchi, and D. Stauch. Aspect-oriented programming for reactive systems: a proposal in the synchronous framework. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 2006. To appear.

[2] K. Altisen, F. Maraninchi, and D. Stauch. Larissa: Modular design of man-machine interfaces with aspects. In *5th International Symposium on Software Composition*, Vienna, Austria, Mar. 2006. To appear.

[3] D. Balzarotti, A. C. D'Ursi, L. Cavallaro, and M. Monga. Slicing AspectJ woven code. In G. T. Leavens, C. Clifton, and R. Lämmel, editors, *Foundations of Aspect-Oriented Languages*, Mar. 2005.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19(2):87–152, 1992.

[5] A. Colyer and A. Clement. Large-scale AOSD for middleware. In K. Lieberherr, editor, *AOSD-2004*, pages 56–65, Mar. 2004.

[6] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In K. Lieberherr, editor, *AOSD-2004*, pages 141–150, Mar. 2004.

[7] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, Sept. 2005.

[8] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology, AMAST'93*, June 1993.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–353, 2001.

[10] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005.

[11] L. Lamport. Proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, SE-3(2):125–143, 1977.

[12] R. E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies*, Mar. 2005.

[13] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001.

[14] R. Pawlak, L. Duchien, and L. Seinturier. Compar: Ensuring safe around advice composition. In *FMOODS 2005*, volume 3535 of *lncs*, pages 163–178, jan 2005.

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

APPENDIX

A. PROOF FOR THEOREM 1

Theorem 1 is a consequence of Theorem 2. We show that

$$advTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_i) \cap$$
$$advTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_{i+1}) = \emptyset$$

follows from

$$advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_i)$$
$$\cap\, advTrans(P_{JP_i} \| P_{JP_{i+1}}, JP_{i+1}) = \emptyset$$

JP_i and JP_{i+1} can only occur in P_{JP_i} and $P_{JP_{i+1}}$. Thus, if a transition that has both of them as outputs in $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$, there must already exist a transition with both of them as outputs in $P_{JP_i} \| P_{JP_{i+1}}$. □

B. PROOF FOR THEOREM 2

Because the parallel product is commutative $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_i} \| P_{JP_{i+1}} \| \ldots \| P_{JP_n})$ and $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_{i+1}} \| P_{JP_i} \| \ldots \| P_{JP_n})$ are the same.

Let $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \triangleleft_{JP_n} \mathrm{adv}_n \ldots \triangleleft_{JP_{i+2}} \mathrm{adv}_{i+2} = (\mathcal{Q}, s_{\mathrm{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T}) = P_{i+2}$. Then $P_{i+2} \triangleleft_{JP_{i+1}} \mathrm{adv}_{i+1}$ yields an automaton $P_{i+1} = (\mathcal{Q}, s_{\mathrm{init}}, \mathcal{I}, \mathcal{O} \cup O_{\mathrm{adv}_{i+1}}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}'$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O\big) \implies$$
$$(s, \ell, O_{\mathrm{adv}_{i+1}}, S\_step_{P'}(s_{\mathrm{init}}, \sigma_{i+1}, l_{\sigma_{i+1}})) \in \mathcal{T}'$$

and $P_{i+1} \lhd_{JP_i} \text{adv}_i$ yields an automaton $P_i = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O} \cup O_{\text{adv}_{i+1}} \cup O_{\text{adv}_i}, \mathcal{T}'')$, where $\mathcal{T}''$ is defined as follows:

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \notin O\big) \implies (s,\ell,O,s') \in \mathcal{T}' \tag{3}$$

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \notin O\big) \implies (s,\ell,O_{\text{adv}_{i+1}}, S\_step_{P'}(s_{\text{init}},\sigma_{i+1},l_{\sigma_{i+1}})) \in \mathcal{T}' \tag{4}$$

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \in O\big) \implies (s,\ell,O_{\text{adv}_i}, S\_step_{P'}(s_{\text{init}},\sigma_i,l_{\sigma_i})) \in \mathcal{T}' \tag{5}$$

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies (s,\ell,O_{\text{adv}_{i+1}}, S\_step_{P'}(s_{\text{init}},\sigma_{i+1},l_{\sigma_{i+1}})) \in \mathcal{T}' \tag{6}$$

If we calculate $P_{i+2} \lhd_{JP_i} \text{adv}_i \lhd_{JP_{i+1}} \text{adv}_{i+1}$, we obtain the same automaton, except for transitions (6), which are defined by

$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies (s,\ell,O_{\text{adv}_i}, S\_step_{P'}(s_{\text{init}},\sigma_i,l_{\sigma_i})) \in \mathcal{T}'\big)$$

Transitions (6) are exactly the join point transitions that are in $\mathit{advTrans}(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_i) \cap \mathit{advTrans}(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_{i+1})$. By precondition, there were no such transitions in $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$. Because we require that all the JP_j outputs occur nowhere else, JP_i and JP_{i+1} cannot be contained in a O_{adv_j}, thus no transition of type (6) has been added by the weaving of $\lhd_{JP_n} \text{adv}_n \ldots \lhd_{JP_{i+2}} \text{adv}_{i+2}$.

Thus, we have $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \lhd_{JP_n} \text{adv}_n \ldots \lhd_{JP_{i+2}} \text{adv}_{i+2} \lhd_{JP_{i+1}} \text{adv}_{i+1} \lhd_{JP_i} \text{adv}_i = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \lhd_{JP_n} \text{adv}_n \ldots \lhd_{JP_{i+2}} \text{adv}_{i+2} \lhd_{JP_i} \text{adv}_i \lhd_{JP_{i+1}} \text{adv}_{i+1}$. Weaving $\lhd_{JP_{i-1}} \text{adv}_{i-1} \ldots \lhd_{JP_1} \text{adv}_1$ trivially yields the same result. □