Formal Analysis Tools for the Synchronous Aspect Language Larissa

David Stauch

Verimag, Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France

Abstract:

We present two tools for the formal analysis of the aspect language Larissa, which extends the simple synchronous language Argos. The first tool concerns the combination of design-by-contract with Larissa aspects, and shows how we can apply an aspect not only to a program, but to a specification of programs in form of a contract, and obtain a new contract. The second concerns aspect interferences, i.e. aspect that influence each other in an unintended way if they are applied to the same program. We present a way to weave aspects in a less conflict-prone manner, and a means to detect remaining conflicts statically. These tools are quite powerful, compared to those available for other aspect languages.

1 Introduction

Aspect-oriented programming (AOP) offers programming constructs to a base language which aim at encapsulating *crosscutting concerns*. These are concerns that cannot be properly captured into a module by the decomposition offered by the base language. AOP languages express crosscutting concerns in *aspects*, and *weave* (i.e. compile) them in the base program with an aspect weaver. All the aspect extensions of existing languages (like AspectJ [13]) share two notions: pointcuts and advice. A *pointcut* describes, with a general property, the program points (called *join points*) where the aspect should intervene (e.g., all methods of the class `X`, or all methods whose names begin with `set`). The *advice* specifies what has to be done at each join point (e.g., execute a piece of code before the normal code of the method).

Reactive systems are control systems that are in constant interaction with their environment. They are often programmed in dedicated languages, which must fulfill specific requirements. First, reactive systems often fulfill safety-critical functions, and thus require the use of formal methods in their development. Programming languages for them must thus be formally defined, and have a connection to verification tools. Furthermore, they usually fulfill several tasks in parallel, and programming languages must thus offer an explicit parallel composition of components.

The family of synchronous languages are such dedicated languages, which are very successfully used to program safety-critical reactive systems, e.g. control systems in airplanes or nuclear power stations. Synchronous languages are all based on the same semantic principle, the synchrony hypothesis, which divides time into instants and assumes that reactions of parallel components are atomic, i.e. that outputs are emitted as soon as the inputs are received. A second principle is the synchronous broadcast, which allows outputs of a component to be read by other components in parallel. These principles allow to develop synchronous languages that are very expressive and have a clear and simple semantics with strong semantic properties. The family of synchronous languages contains languages with different styles, e.g. the dataflow language Lustre [11] and the imperative language Esterel [4]. The simplest language of the family is Argos [20], a hierarchical automata language, based on Mealy machines, which can be composed by different operators.

There are also crosscutting concerns in synchronous programs, which cannot be encapsulated with the parallel composition and other operators of synchronous languages. They are however different from crosscutting concerns in programs written in general-purpose languages, because they crosscut the *parallel* structure of reactive programs. Therefore, and because they are usually not formally defined, existing aspect languages cannot be applied to reactive systems. Thus, we developed an aspect-oriented extension for Argos, called Larissa [1].

When designing Larissa, we took great care to give it a clean and simple semantics and strong semantic properties, as they are common in synchronous languages. Thus, pointcuts are specified as synchronous *observers* [12], i.e. Argos programs that, via the synchronous broadcast, observe the inputs and the outputs of the base program, and compute a safety property on them. This is a semantic and at the same time

very expressive mechanism. Larissa has different kinds of advice, and all are specified depending only on the interface of the base program, but not on its implementation. Due to this semantic definition, Larissa aspects preserve the equivalence of base programs.

Having a clean and simple semantics has the advantage of making programs easier to understand for programmers. Furthermore, it allows the semantic analysis of programs. In this paper, we present two tools for semantic analysis.

The first combines Larissa aspects with another successful programming technique, Design-by-Contract [21], which has been originally introduced for object-oriented systems. There, a method is specified by a contract, which consists of an *assumption* clause and a *guarantee* clause. It fulfills its contract if after its execution, the guarantee holds whenever the assumption was true when the program was called.

Contracts have been adapted to reactive systems by [19], where assumptions and guarantees are expressed as observers, in the same way as Larissa pointcuts. Because reactive systems constantly receive inputs and emit outputs, it seems natural to let the assumption observer restrict the inputs, and let the guarantee observer ensure properties on the outputs.

Aspect-oriented programming and design-by-contract can hardly be used concurrently: when an aspect is applied to a method, it changes its semantics, such that its contract is no longer valid. The approach we present solves this problem for Argos and Larissa by generating a new contract that is valid after the application of the aspect. We show how to apply an aspect asp to a contract C and derive a new contract C', such that for any program P which fulfills C, P with asp fulfills C'. Although an observer is also an Argos program, we cannot directly apply aspects, because it has a different interface, where the outputs of the program have become inputs. We therefore transform the observers first into non-deterministic Argos programs, which produce exactly the execution traces the observer accepted, and apply the aspect to these. A second difficulty comes from the fact that we must treat assumption and guarantee differently to preserve the correctness of our algorithm. We demonstrate this approach on an example which models a tramway door controller.

The second semantic analysis we present treats interference between aspects. Applying several aspects to the same program may lead to unintended results because of conflicts between the aspects. We say that two aspects *interfere* when weaving them in different orders does not yield the same result.

Whether two aspects interfere depends on the way they are woven in the program. We distinguish *sequential* and *joint* weaving. Sequential weaving means weaving the aspects one by one, where the next aspect is woven in the result of the previous weaving. Argos operators are defined that way, and also Larissa aspects. On the other hand, joint weaving means weaving several aspects together, into the same base program. AspectJ is defined that way: its semantics is not defined as a transformation of the base program, but as injecting behavior in the running program, including other aspects.

Sequential weaving often causes interference between aspects, because the second aspects is applied to the first, but not the other way round. Therefore, we present a joint weaving mechanism for Larissa, which applies aspects to the same base program, and thus reduces interferences. As opposed to AspectJ, however, all jointly woven aspects only affect the base program, but not each other. Therefore, we still need sequential weaving, in cases where one aspect needs to affect another.

Joint weaving removes many cases of interference, which we also demonstrate with an example. However, interference is unavoidable when two aspects want to modify the base program in the same point. Such cases should be made explicit to the programmer. We therefore present a interference analysis for jointly woven aspects, that can either determine that two aspects do not interfere for a given base program, or that they never interfere for any base program. In the first case, we apply both pointcuts to the base program and check if there are common join points. In the second case, it is sufficient to perform a parallel product of the two pointcuts. All these steps must be performed during the compilation process anyway, and thus add no additional cost.

Both tools we present in this paper are only possible because of the semantic definition of Larissa. Thus, the contract weaving can apply aspects to programs whose implementation is unknown. The interference analysis also depends on the semantic definition of Larissa, notably on the precise description of join points with observers, which makes it possible to determine statically the points where several aspects want to introduce their advice.

The structure of the paper is as follows: Section 2 introduces Larissa and Argos, Section 3 shows how to weave contracts in aspects, and Section 4 contains an example for this. Next, Section 5 explains the inter-

ference analysis, using a second example. Section 6 discusses related work, and Section 7 concludes. Work on the combination of contracts and aspects has been published in [25], and work on aspect interference in [26].

2 Argos and Larissa

This section presents a restriction of the Argos language [20], and the Larissa extension [1]. Argos 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 finite sets of Boolean input and output variables representing signals from and to the environment. An* input trace*, it, is a function: $it : N \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. An* output trace*, ot, is a function: $ot : N \longrightarrow [\mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. 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 $n \in N$. 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 $n \in N$.*

A set of pairs of i/o-traces $S = \{(it, ot) \mid it \in InputTraces \wedge ot \in OutputTraces\}$ is deterministic *iff $\forall (it, ot), (it', ot') \in S \ . \ (it = it') \implies (ot = ot')$, and it 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, and the encapsulation is the operator that expresses the communication between automata. 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.*

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:

$$S_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times N \longrightarrow 2^{\mathcal{Q}}$$
$$O_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times N \setminus \{0\} \longrightarrow 2^{(2^{\mathcal{O}})}$$

$S_step(s, it, n)$ returns the set of states that are reachable from state s after performing n steps with the input trace it; $O_step(s, it, n)$ contains the different combinations of outputs that can be emitted at step n

after executing it from s:

$$\begin{aligned} n = 0: \ & S_step_{\mathcal{A}}(s, it, n) = \{s\} \\ n > 0: \ & s' \in S_step_{\mathcal{A}}(s, it, n) \qquad O \in O_step_{\mathcal{A}}(s, it, n) \\ & \text{where } s'' \in S_step_{\mathcal{A}}(s, it, n-1) \wedge \exists (s'', \ell, O, s') \in \mathcal{T} \\ & \wedge\, \ell \text{ has value true for } it(n-1)\ . \end{aligned}$$

Note that if the automaton is deterministic and complete, S_step and O_step always return a set with a single element.

We denote *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*$(\mathcal{A}_2)$. We assume that Argos programs are deterministic and complete, as these are important properties for reactive systems.

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* $\mathcal{A} \backslash \Gamma = (\mathcal{Q}, s_{init}, \mathcal{I} \backslash \Gamma, \mathcal{O} \backslash \Gamma, \mathcal{T}')$ *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. $\ell^- = \{x \in \mathcal{I} \mid (\overline{x} \wedge \ell) = \ell\}$). $\exists \Gamma\ .\ \ell$ is then defined as $\exists \Gamma\ .\ \ell = \bigwedge_{a \in \ell^+ \backslash \Gamma} a \wedge \bigwedge_{a \in \ell^- \backslash \Gamma} \overline{a}$.

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)).

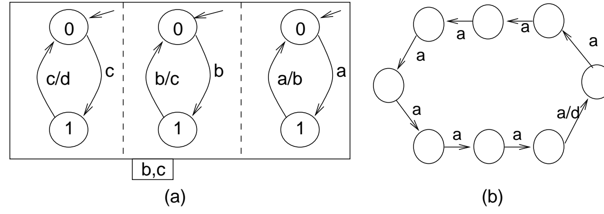


Figure 1: A 3-bits counter. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering condition / emitted outputs`", e.g. the transition labelled by "`a/b`" is triggered when `a` is true and emits `b`.

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: a 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.

The goal of aspects being to specify such cross-cutting modifications, we proposed an aspect-oriented extension for Argos [1], which allows the modularization of a number of recurrent problems in reactive programs, like the reinitialization. This leads to the definition of a new operator (the aspect weaving operator), which preserves determinism and completeness of programs, as well as semantic equivalence between programs. Similar to aspects in other languages, a Larissa aspect consists of a pointcut, which selects a set of join points, and an advice, which modifies these join points.

2.3.1 Join Point Selection

To preserve semantical equivalence, pointcuts in Larissa are not expressed in terms of the internal structure of the base program (as e.g. state names), but 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* [12] are a powerful and well-understood mechanism which may be used to describe pointcuts. Indeed, an observer is a program that may observe the inputs and the outputs of the base program, without modifying its behavior, and computes a safety property (in the sense of safety/liveness properties as defined in [14]).

Therefore, observers are well suited to express pointcuts. A pointcut is thus an observer which selects a set of *join point transitions* by emitting a single output *JP*, the *join point signal*. A transition T in a program P is selected as a join point transition when in the concurrent execution of P and the pointcut, *JP* is emitted when T is taken.

Technically, we perform a synchronous product between the program and the pointcut and select those transitions in the product which emit *JP*. However, if we simply put a program P and an observer PC in parallel, P's outputs $\mathcal{O}$ will become synchronization signals between them, as they are also inputs of PC. 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 PC, and o will still be visible as an output. First, we transform P into P' and PC into PC$'$, 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:

$$\boldsymbol{\mathcal{P}}(P, \mathrm{PC}) = (P' \| \mathrm{PC}' \| dupl_{o_1} \| \; ... \; \| dupl_{o_n}) \setminus \{o'_1, ..., o'_n\}$$

The join point transitions are those transitions of $\boldsymbol{\mathcal{P}}(P, \mathrm{PC})$ that emit *JP*. Fig. 2 illustrates the pointcut mechanism. The pointcut (b) specifies any transition which emits `c`: in base program (a), the loop transition in state B is selected as a join point transition.

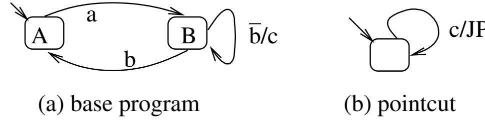


Figure 2: Example pointcut.

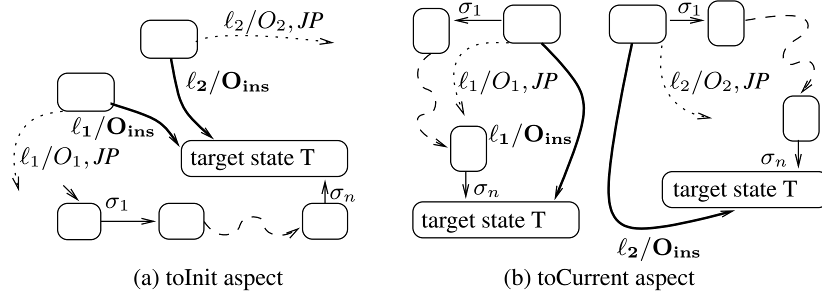


Figure 3: Schematic toInit and toCurrent aspects. Advice transitions are in bold, join point transitions are dotted.

2.3.2 Specifying the Advice

In aspect oriented languages, the advice expresses the modification applied to the base program. In Larissa, we define two types of advice: in the first type, an advice replaces the join point transitions with *advice transitions* pointing to an existing target states; in the second type, an advice introduces a Argos program between the source state of the join point transition and an existing target state. In both cases, target states have to be specified without referring explicitly to state names.

An advice *adv* has two ways of specifying the target state T among the existing states of the base program P. T is the state of P that would be reached by executing a finite input trace from either the initial state of P, *adv* is then called toInit advice, or from the source state of the join point transition, *adv* is then called toCurrent advice. As the base program is deterministic and complete, executing an input trace from any of its states defines exactly *one* state.

The advice weaving operator$\triangleleft_{JP} adv$ weaves a piece of advice *adv* in a program.

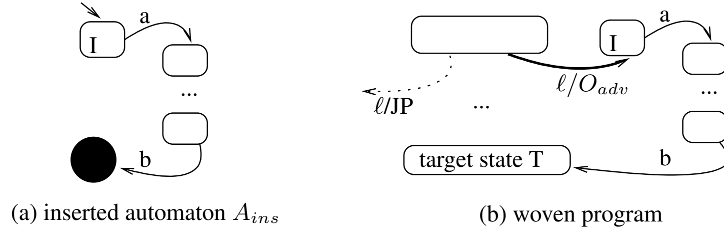
Advice Transitions. The first type of advice consists in replacing each join point transition with an advice transition. Once the target state is specified by a finite input trace $\sigma = \sigma_1 \dots \sigma_n$, the only missing information is the label of these new transitions. We do not change the input part of the label, so as to keep the woven automaton deterministic and complete, but we replace the output part by some *advice outputs* O_{adv}. These are the same for every advice transition, and are thus specified in the aspect. Advice transitions are illustrated in Fig. 3.

Formal Definition. We only define toInit advice formally. A formal definition of the complete Larissa language can be found in [24, Chapter 4].

Definition 5 (toInit advice weaving)**.** *Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an deterministic and complete automaton and $adv = (O_{adv}, toInit, \sigma)$ a piece of advice, with $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ 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:*

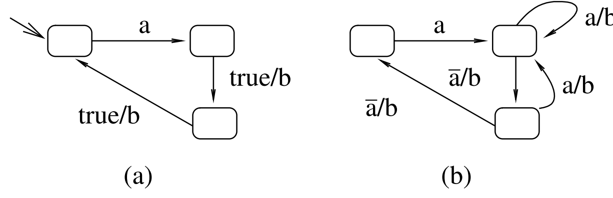
$$\big((s,\ell,O,s') \in \mathcal{T} \wedge JP \notin O\big) \implies (s,\ell,O,s') \in \mathcal{T}' \quad (1)$$

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



(a) inserted automaton A_{ins} (b) woven program

Figure 4: Inserting an advice automaton.



(a) (b)

Figure 5: A mono stable flip-flop (a), made retriggerable (b).

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 $\mathcal{A}$.

Advice Programs. It is sometimes not sufficient to modify single transitions, i.e. to jump to another location in the automaton in only one step. It may be necessary to execute arbitrary code when an aspect is activated. In these cases, we can insert an automaton between the join point and the target state.

Therefore, we use an *inserted automaton* A_{ins} that can *terminate*. Since Argos has no built-in notion of termination, the programmer of the aspect has to identify a final state F (denoted by filled black circles in the figures).

We first specify a target state T as explained above. Then, for every T, a copy of the automaton A_{ins} is inserted, which means: 1) replace every join point transition J with target state T by a transition to the initial state I of this instance of A_{ins}. As for advice transitions, the input part of the label is unchanged and the output part is replaced by O_{adv}; 2) connect the transitions that went to the final state F in A_{ins} to T. Advice programs are illustrated in Fig. 4.

2.3.3 Fully Specifying an Aspect

An aspect is given by the specification of its pointcut and its advice: $asp = (\text{PC}, adv)$, where PC is the pointcut and *adv* is the advice. *adv* is a tuple which contains 1) the advice outputs O_{adv}; 2) the *type* of the target state specification (*toInit* or *toCurrent*); 3) the finite trace σ over the inputs of the program; and optionally, 4) P_{adv}, the advice program. Thus, advice can be a tuple $< O_{adv}, type, \sigma >$, or, with an advice program, a tuple $< O_{adv}, type, \sigma, P_{adv} >$, with *type* $\in \{toCurrent, toInit\}$. An aspect is woven into a program by first determining the join point transitions and then weaving the advice.

Definition 6 (Aspect weaving)**.** *Let P be a program and $asp = (PC, adv)$ an aspect for P. The weaving of asp on P is defined by*

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

2.3.4 Example

As an example, consider a mono-stable flip-flop (MFF) with one input `a` and one output `b`, which emits two `b`s after it received an `a`. Fig. 5(a) shows an implementation of the MFF in Argos. We want to make the MFF retriggerable, meaning that if an `a` is emitted during several following instants, the MFF

continues emitting b. We do this by applying the aspect $A_{\text{tri}} = (\text{PC}, < b, \text{toInit}, (a) >)$ to the MFF, where PC =({S},S,{a,b},{*JP*}, {(S,a.b,*JP*,S)}) is a pointcut which selects all occurrences of a.b as join points. Fig. 5(b) shows the result of applying A_{tri} to the implementation.

3 Combining Contracts and Aspects

In this section, we show how to apply aspects to a specification of programs in form of a contract. First, we formally define contracts for Argos, then explain informally how to weave aspects into them, and finally define this process formally.

3.1 Contracts for Argos

The observers we use in contracts are slightly different from those used as pointcuts. Notably, once they start emitting their output err, they continue emitting it forever. This is done in an *Error State* Error. Such an observer specifies a class of programs fulfilling a certain safety property, namely those programs where the observer never emits err when combined with them. The Error State is thus a way to refuse certain inputs while keeping the observer complete.

Definition 7 (Observer)**.** *An observer is an automaton* $(\mathcal{Q} \cup \{\mathit{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ *which observes an automaton with inputs* $\mathcal{I}$ *and outputs* $\mathcal{O}$*. When an observer emits* err*, it will go to state Error and also emit* err *in the next instant. A program* P *is said to* obey *an observer obs (noted* $P \models \mathit{obs}$*) iff* $P \| \mathit{obs} \setminus \mathcal{O}$ *produces no trace which emits* err*.*

Transitions leading to the Error State are called *Error transitions*.

A contract specifies a class of programs with two observers, an assumption and a guarantee. Definition 8 is an auxiliary definition, used to formally define contracts in Definition 9. $\diamond$ denotes the trace for a single output err that never emits err, i.e. $\diamond(\texttt{err})[n] = \texttt{false}$ for all n. An observer that accepts a trace emits $\diamond$.

Definition 8 (Trace Combination)**.** *Let* $it : N \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *and* $ot : N \longrightarrow [\mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *be traces, with* $\mathcal{I} \cap \mathcal{O} = \emptyset$*. Then,* $it.ot : N \longrightarrow [\mathcal{I} \cup \mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *is a trace s.t.* $\forall i \in \mathcal{I} \,.\, it.ot(n)(i) = it(n)(i) \wedge \forall o \in \mathcal{O} \,.\, it.ot(n)(o) = ot(n)(o)$*.*

Definition 9 (Contract)**.** *A contract over inputs* $\mathcal{I}$ *and outputs* $\mathcal{O}$ *is a tuple (A,G) of two observers over* $\mathcal{I} \cup \mathcal{O}$*, where A is the assumption and G is the guarantee. A program P fulfills a contract (A,G), written* $P \models$ *(A,G), iff*

$$(it.ot, \diamond) \in \mathit{Traces}(A) \wedge (it, ot) \in \mathit{Traces}(P) \Rightarrow (it.ot, \diamond) \in \mathit{Traces}(G) \ .$$

Intuitively, a guarantee G should only restrict the outputs of a program and an assumption A should only restrict the inputs. We do not require this formally, but contracts which do not respect this constraint are of little use. Indeed, if G restricts the inputs more than A, it follows from Definition 9 that there exists no program P s.t. $P \models (A,G)$. Conversely, a program is usually placed in an environment E, s.t. $E \models A$. If A restricts the outputs, no such E exists, as the outputs are controlled by P.

As an example for a contract, consider the following contract for the MFF from Section 2.3.4. The contract is composed of an assumption, shown in Fig. 6(a), which states that a's always occur in pairs, and a guarantee consisting of two automata, shown in Fig. 6(b) and (c), which are composed in parallel. The automaton in Fig. 6(b) guarantees that a single b is never emitted, and the automaton in Fig. 6(c) guarantees that when a occurs while no b is emitted, b is emitted in the next instant. The product of Fig. 6(b) and Fig. 6(c) is shown in Fig. 6(d).

3.2 Weaving Aspects in Contracts

We want to apply an aspect asp not to a specific program P, but to a class of programs defined by a contract C, and obtain a new class of programs, defined by a contract C', such that $P \models C \Rightarrow P \triangleleft \, asp \models C'$. To

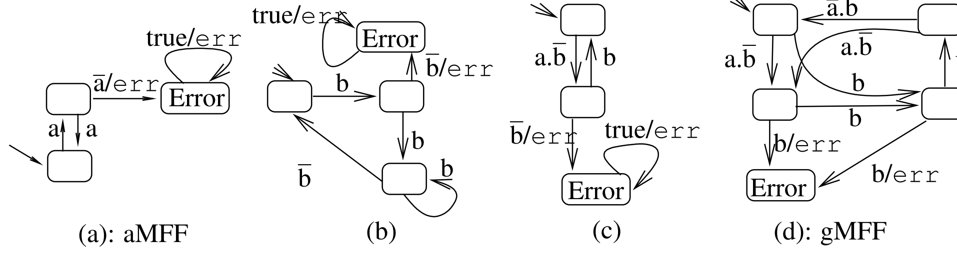


Figure 6: The contract for the MFF.

construct C', we simulate the effect that the aspect has on a program as far as possible on the assumption and the guarantee observers of C. However, an aspect cannot be applied directly to an observer, because the aspect has been written for a program with inputs $\mathcal{I}$ and outputs $\mathcal{O}$, whereas for the observer, $\mathcal{O}$ are also inputs.

Therefore, we transform the observers of the contract first into non-deterministic automata (NDA), which produce exactly those traces that the observer accepts. We then weave the aspects into the NDA, with a modified definition of the weaving operator. The woven NDA are then transformed back into observers. The obtained observers may still be non-deterministic, and are thus determinized.

Except for aspect weaving, all of these steps are different for the assumption and the guarantee, as far as the Error transitions are concerned. This is because the assumption and the guarantee have different functions in a contract: the assumption states which part of the program is defined by the contract, and the guarantee gives properties that are always true for this part. Indeed, a contract (A,G) can be rewritten as (`true`,$A \Rightarrow G$), where $A \Rightarrow G$ is an observer that emits `err` when G emits `err` but not A. Thus, the assumption can be considered as a negated guarantee.

After weaving an aspect, the assumption must exclude the undefined part of *any* program which fulfills the contract. Therefore, it must reject a trace (by emitting `err`) as soon as there exists a program for which it cannot predict the behavior. The guarantee, on the other hand, emits `err` only for traces which cannot be emitted by any program which fulfills the contract. Therefore, after weaving an aspect, the new guarantee may only emit `err` if it is sure that there exists no program that produces the trace.

On the other hand, we want the assumption to be as permissive as possible, to include all possible programs, and the guarantee as restrictive as possible, to characterize the woven program as exactly as possible. Thus, when we know exactly the behavior of the program, as e.g. that of an inserted advice program, we do not emit `err` in the assumption, but we emit `err` in the guarantee to exclude all input/output combinations that are never produced by the program.

3.3 Formal Definitions

This section describes the weaving of aspects into contracts in detail, and illustrates it on the MFF example. First, Definition 10 defines the transformation of an observer into a NDA through two functions, one for guarantee observers and one for assumption observers.

Definition 10 (Observer to NDA transformation)**.** *Let obs* $= (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ *be an observer with an error state Error over inputs* $\mathcal{I}$ *and outputs* $\mathcal{O}$*, with* $\mathcal{I} \cap \mathcal{O} = \emptyset$*.* $ND_G(obs) = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_G})$ *defines a NDA, where* T_{ND_G} *is defined by* $(s, \ell_{\mathcal{I}} \wedge \ell_{\mathcal{O}}, \emptyset, s') \in T \Rightarrow (s, \ell_{\mathcal{I}}, \ell_{\mathcal{O}}^{+}, s') \in T_{ND_G}$*.* $ND_A(obs) = (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_A})$ *defines a NDA, where* T_{ND_A} *is defined by* $(s, \ell_{\mathcal{I}} \wedge \ell_{\mathcal{O}}, o, s') \in T \Rightarrow (s, \ell_{\mathcal{I}}, \ell_{\mathcal{O}}^{+} \cup o, s') \in T_{ND_G}$*.*

Note that the transitions in *obs* which emit `err` (i.e. the Error transitions) have no corresponding transitions in $ND_G(obs)$. In the guarantee, these transitions correspond to input/output combinations which are never produced by the program and must not be considered by the aspect. The other transitions are transformed such that part of the condition concerning $\mathcal{O}$ disappears, and those outputs that appeared as positive atoms in the condition (i.e., $\ell_{\mathcal{O}}^{+}$) become outputs.

As an example, consider the guarantee of the MFF (Fig. 6(d)). Its transformation into a NDA is shown in Figure 7(a). Note that the Error State and the transitions leading to it have disappeared, and that b is now an output. Thus, the transition label b has been transformed to true/b, and label a.$\overline{\text{b}}$ to a.

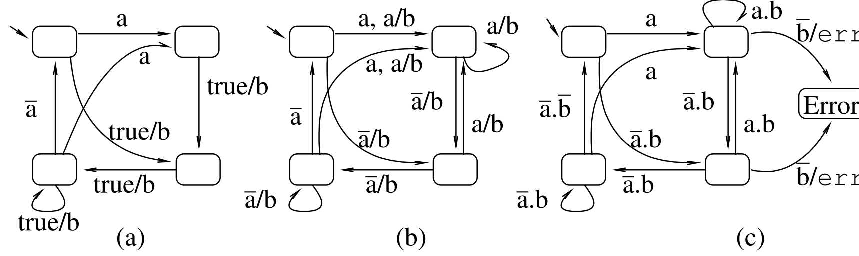


Figure 7: a: ND_G(gMFF), b: ND_G(gMFF)$\triangleleft$ A_{tri}, c: OBS_G(ND_G(gMFF)$\triangleleft$ A_{tri}).

In the assumption, on the other hand, the Error transition correspond to inputs from the environment to which the program may react arbitrarily. If the aspect replaces these transitions in the assumption, they are also replaced in the program, and can thus be accepted from the environment by the woven program. Thus, error transitions are not removed in $ND_A(obs)$, so that the aspect weaving can modify them. The transformation of the assumption of the MFF (Fig. 6(a)) is shown in Fig. 8(a).

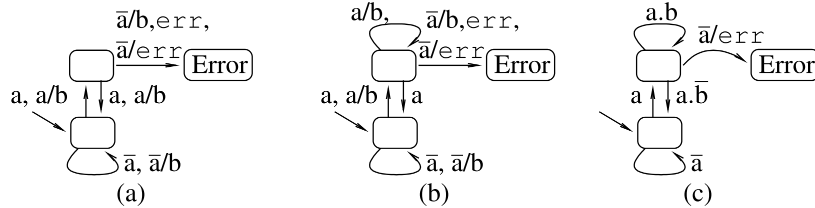


Figure 8: a: ND_A(aMFF), b: ND_A(aMFF)$\triangleleft$ A_{tri}, c: OBS_A(ND_A(aMFF)$\triangleleft$ A_{tri}).

We can now apply an aspect to a NDA. However, a trace may lead to several states. Thus, for each join point transition, several advice transitions must be created, one for each target state. We only give a definition for toInit advice, but the extension to toCurrent advice and advice programs is straightforward, and can be found in [24, Chapter 8].

Definition 11 (toInit weaving for NDA)**.** *Let* $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $adv = (O_{adv},$ *toInit*$, \sigma)$ *a piece of toInit advice, with* $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *a finite input trace of length* $\ell_\sigma + 1$*. The advice weaving operator* $\triangleleft$*, weaves adv into* $\mathcal{A}$ *and returns the automaton* $\mathcal{A} \triangleleft adv = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O} \cup O_{adv}, \mathcal{T}')$*, where* $\mathcal{T}'$ *is defined as follows:*

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \tag{3}$$

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\big) \implies \forall targ \in S_step_{\mathcal{A}}(s_{init}, \sigma, \ell_\sigma) \,.\, (s, \ell, O_{adv}, targ) \in \mathcal{T}' \tag{4}$$

Transitions (3) are not join point transitions and are left unchanged. Transitions (4) 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 $\mathcal{A}$. Fig. 7(b) and Fig. 8(b) show the NDAs from our example with the retriggerable aspect from Section 2.3.4 woven into them. For both NDAs, the trace leads to a single state, thus only one advice transition is introduced per join point transition.

Transforming a NDA back into an observer is different for assumptions and guarantees. In the assumption, we do not add additional error transitions, but only leave those already there. In the guarantee, we add transitions to the error state from every state where the automaton is not complete. This is correct, as these transitions correspond to traces that are never produced by any program.

Definition 12 (NDA to guarantee transformation)**.** *Let* $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T)$ *be a NDA.* $OBS_G(nd) = (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\mathtt{err}\}, T' \cup T'')$ *defines an observer, where* T' *and* T'' *are defined by*

$$(s, \ell, o, s') \in T \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O} \setminus o}}, \emptyset, s') \in T' \tag{5}$$

$$\begin{aligned}&(s, \ell, \emptyset, s') \notin T' \wedge s \in \mathcal{Q} \wedge \ell \text{ is a complete monomial over } \mathcal{I} \cup \mathcal{O}\\ &\qquad \Rightarrow (s, \ell, \{\mathtt{err}\}, Error) \in T''\end{aligned} \tag{6}$$

where $l_O = \bigwedge_{o \in O} o$ *and* $l_{\overline{O}} = \bigwedge_{o \in O} \overline{o}$ *for a set* O *of variables.*

When transforming an NDA to an assumption, we do not add additional error transitions, but only leave those already there.

Definition 13 (NDA to assumption transformation)**.** *Let* $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O} \cup \{\mathtt{err}\}, T)$ *be a NDA.* $OBS_A(nd) = (\mathcal{Q}, q_0, \mathcal{I} \cup \mathcal{O}, \{\mathtt{err}\}, T')$ *defines an observer, where* T' *is defined by*

$$(s, \ell, o \cup e, s') \in T \wedge o \subseteq \mathcal{O} \wedge e \subseteq \{\mathtt{err}\} \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O} \setminus o}}, e, s') \in T'$$

Fig. 7(c) and Fig. 8(c) show the NDAs from our example transformed back into observers. As expected, the obtained guarantee in Fig. 7(c) tells us that whenever the program receives an `a`, it emits `b`'s the two following instants. The assumption, however, requires that if an `a` is emitted, it continues to be emitted until there is no `b`.

The resulting observer may not be deterministic. However, it can be made deterministic, as observers are acceptor automata. Determinization for guarantees and assumptions is different: a guarantee must only emit `err` for a trace σ if all programs fulfilling the contract never emit σ, and an assumption must emit `err` if there exists a program fulfilling the contract which is not defined for σ.

Existing determinization algorithms can be easily adapted to fulfill these requirements. We do not detail such algorithms here, but instead give conditions the determinization for assumptions and guarantees must fulfill. The new assumption and the new guarantee in the example are already deterministic, thus there is no need to determinize them.

The assumption determinization gives precedence to error transition. If there is a choice between an error transition and a non-error transition, the error transition is always taken. Thus, the determinized assumption only accepts a program if all possible non-deterministic executions of the non-determinized assumption accept it.

Definition 14 (Assumption Determinization)**.** *Let* M *be a NDA with outputs* $\{\mathtt{err}\}$*.* $Det_A(M)$ *is a deterministic automaton such that*

$$\begin{aligned}&(it, ot) \in Traces(Det_A(M)) \Leftrightarrow\\ &(it, ot) \in Traces(M) \wedge \nexists ot' \,.\, (it, ot') \in Traces(M) \wedge ot'(n)[\mathtt{err}] = \mathtt{true} \wedge ot(n)[\mathtt{err}] = \mathtt{false}\ .\end{aligned}$$

As opposed to the assumption determinization, the guarantee determinization gives precedence to non-error transitions over error transitions. Thus. the determinized guarantee emits `err` only if all possible executions of the non-determinized guarantee also emit `err`.

Definition 15 (Guarantee Determinization)**.** *Let* M *be a NDA with outputs* $\{\mathtt{err}\}$*.* $Det_G(M)$ *is a deterministic automaton such that*

$$\begin{aligned}&(it, ot) \in Traces(Det_G(M)) \Leftrightarrow\\ &(it, ot) \in Traces(M) \wedge \nexists ot' \,.\, (it, ot') \in Traces(M) \wedge ot'(n)[\mathtt{err}] = \mathtt{false} \wedge ot(n)[\mathtt{err}] = \mathtt{true}\ .\end{aligned}$$

We can now state the following theorem, which states that a contract constructed with the above operations holds indeed for any program fulfilling the original contract with an aspect applied to it.

Theorem 1. *Let* P *be a program and let* (A, G) *be a contract. Then,*

$$P \models (A, G) \Rightarrow P \triangleleft asp \models (Det_A(OBS_A(ND_A(A) \triangleleft asp)), Det_G(OBS_G(ND_G(G) \triangleleft asp)))\ .$$

Theorem 1 first transforms the assumption and the guarantee into NDA with the respective operators, then applies the aspect to both and transforms the result back in observers, which are determinized. We prove it in Appendix A.

Controller Inputs: / **Controller Outputs:**

Controller Inputs		Controller Outputs	
inStation	Tram is in station	doorOK	door is closed and ready to leave
leaving	Tram wants to leave station	openDoor	opens the door
doorOpen	the door is open	closeDoor	closes the door
doorClosed	the door is closed	beep	emits a warning sound
askForDoor	a passenger wants to leave the tram	setTimer	starts a timer
timer	the timer set by setTimer has run out		

Gangway Inputs		Gangway Outputs	
gwOut	the gangway is fully extended	extendGW	extends the gangway
gwIn	the gangway is fully retracted	retractGW	retracts the gangway
askForGW	a passenger wants to use the gangway		

Helper Signals Outputs	
acceptReq	the passenger can ask for the door or the gw
doorReq	the passenger has asked for the door to open
gwReq	the passenger has asked for the gangway
depImm	the tramway wants to leave the station

Figure 9: The interfaces of the controller and the gangway, and the helper signals.

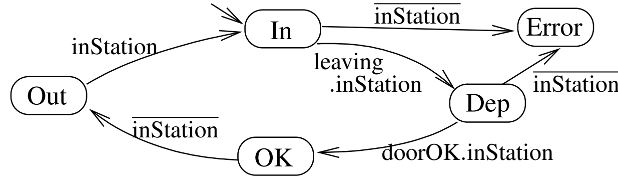


Figure 10: Model of the tramway, M_{Tram}.

4 Example: The Tramway Door Controller

We implement and verify a larger example, taken from the Lustre tutorial [18], a controller of the door of a tramway. The door controller is responsible for opening the door when the tram stops and a passenger wants to leave the tram, and for closing the door when the tram wants to leave the station. Doors may also include a gangway, which can be extended to allow passengers in wheelchairs enter and leave the tram.

We implement the controller as an Argos program. We first develop a controller for a door without the gangway, and then add the gangway part with aspects. Fig. 9 gives the in- and outputs of the controller with their specifications, and also the in- and outputs which are added by the gangway. The controller uses additional inputs, called Helper Signals, which are also shown in Fig. 9 and are calculated from the original inputs.

It is important for the safety of the passengers that the doors are never open outside a station. We call this property P_{Safety}, and formally express it as an observer that emits `err` whenever $\overline{\text{doorClosed}}.\overline{\text{inStation}}$ is true. to formally verify this property, we must first develop a model that describes the possible behavior of the physical environment of the controller, which consists of the door and the tramway. These models are also expressed as Argos observers. The models for the tramway (called M_{Tram}) and the door (called M_{Door}) are shown in Fig. 10 and Fig. 11 respectively. These models require that the environment behaves correctly (e.g., the door only opens if `openDoor` has been emitted).

Furthermore, we give a contract for the door controller, which focuses on P_{Safety}. The guarantee G_{Contr} of the contract is shown in Fig. 12, it ensures that the controller emits `doorOK` only if the doors are closed, and `openDoor` only if the tram is in a station. The contract has also an assumption A_{Contr}, which is the model of the door given in Fig. 11, i.e. $A_{\text{Contr}} = M_{\text{Door}}$. An implementation I_{Contr} of the controller, which fulfills the contract, is given in Fig. 13.

We can now prove that the controller satisfies the contract ($I_{\text{Contr}} \models (A_{\text{Contr}}, G_{\text{Contr}})$), and that the contract in the environment never violates the safety property. Formally, this is expressed as $M_{\text{Door}} \parallel M_{\text{Tram}} \parallel G_{\text{Contr}} \models P_{\text{Safety}}$, where the synchronous product of observers means that the properties expressed by all the observers must be fulfilled.

Adding The Gangway. Two aspects are used to add support for the gangway: one aspect A_{ext} that extends the gangway before the door is opened if a passenger has asked for it, and one aspect A_{ret} that retracts the gangway when the tram is about to leave, if it is extended.

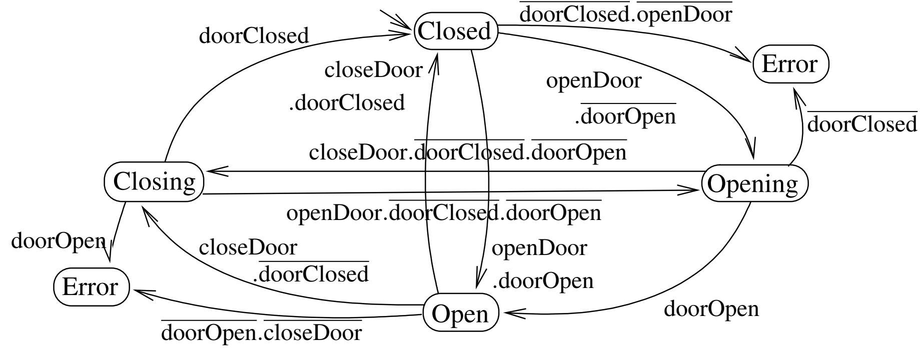


Figure 11: Model of the door, M_{Door}.

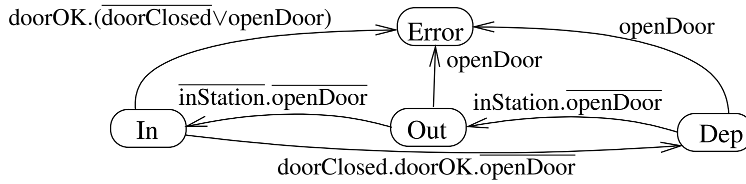


Figure 12: The guarantee of the contract of the controller, G_{Contr}.

The pointcut PC_{ext} of A_{ext} selects all transitions where $openDoor.doorReq.doorClosed.\overline{gwOut}$ is true, and the pointcut PC_{ret} of A_{ret} selects all transitions where $doorOK.\overline{gwIn}$ is true.

Both aspects insert an automaton and return then to the initial state of the join point transitions. The inserted automata for the aspects are shown in Fig. 14. A_{ext} is specified by $(\text{PC}_{\text{ext}}, < \emptyset, \mathit{toCurrent}, (), I_{\text{ext}} >)$, and A_{ret} by $(\text{PC}_{\text{ret}}, < \{retractGW\}, \mathit{toCurrent}, (), I_{\text{ret}} >)$. Weaving these aspects into I_{Contr} adds one state between `Closed` and `OK`, where the gangway is retracted, and one state before `Opening`, where it is extended.

Modularly Verifying the Safety Properties. We want to check that the new controller $I_{\text{Contr}} \triangleleft A_{\text{ext}} \triangleleft A_{\text{ret}}$ still verifies the safety property from above, and also verifies two new safety properties, which require that the gangway is always fully retracted while the tram is out of station, and that the gangway is never moved when the door is not closed. We express these three properties as an observer and call it P_{Safeties}. To verify this, we first weave the aspects into the contract, and thus obtain a new contract that holds for controller with the aspects. Then, we check then that the environment, to which we added a model of the gangway M_{GW}, satisfies the new assumption (i.e., $M_{\text{Door}} \parallel M_{\text{Tram}} \parallel M_{\text{GW}} \models A_{\text{Contr}} \triangleleft A_{\text{ext}} \triangleleft A_{\text{ret}}$), and that the new

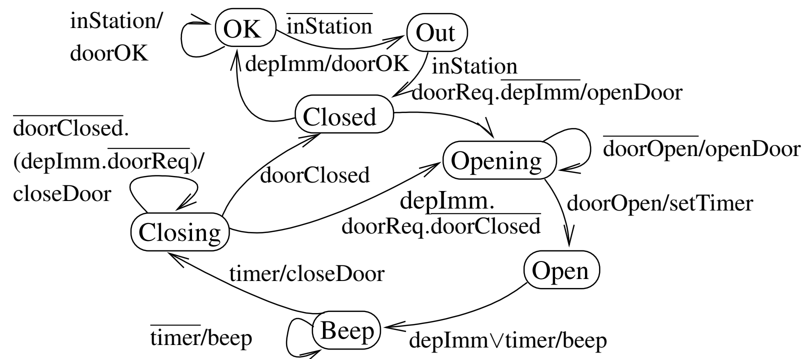


Figure 13: A sample controller for the tramway door, I_{Contr}.

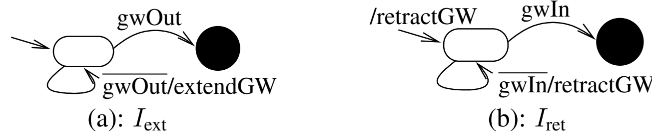


Figure 14: Inserted automata for A_{ext} (a) and A_{ret} (b).

guarantee satisfies the safety requirements in the environment (i.e., $M_{\text{Door}} \parallel M_{\text{Tram}} \parallel M_{\text{GW}} \parallel G_{\text{Contr}} \lhd A_{\text{ext}} \lhd A_{\text{ret}} \models P_{\text{Safeties}}$).

An alternative to this modular approach is to verify directly that the sample controller with the aspects does not violate the given safety properties (i.e., $M_{\text{Door}} \parallel M_{\text{Tram}} \parallel M_{\text{GW}} \parallel I_{\text{Contr}} \lhd A_{\text{ext}} \lhd A_{\text{ret}} \models P_{\text{Safeties}}$). One disadvantage of the alternative approach is that the woven controller may be much bigger than the woven contract. To illustrate this problem, we verified the safety properties using our implementation [15]. The source code of the door controller example is available at [16]. Verifying the woven program takes 11.0 seconds[1]. On the other hand, weaving the aspects into the guarantee of the controller contract and verifying against the environment takes 3.7 seconds[1], and verifying that the sample controller verifies the contract and verifying that the environment fulfills the assumption with the aspects takes < 0.5 seconds[1]. Thus, using this modular approach to verify the safety properties of the controller is significantly faster than verifying the complete program. Although the size of the woven controller is not prohibitive in this example, this indicates that larger programs can be verified using the modular approach.

5 Aspect Interference

A key point when dealing with aspects is the notion of interferences, which is closely related to the way aspects are woven. We illustrate the problem of interfering aspects on an example presented in Section 5.1. Next, we also present a new weaving algorithm in Section 5.2, that weaves aspects *jointly*, and removes aspect interferences in many cases. Finally, we introduce an algorithm in Section 5.3 that proves non-interference of aspects or identifies remaining interferences in jointly woven programs.

5.1 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[2] model by Suunto[2].

5.1.1 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 Fig. 15. 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 [20] for a definition). We do not present this 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 5.2 is also the same.

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

[1]Experiments were conducted on an Intel Pentium 4 with 2.4GHz and 1 Gigabyte RAM.
[2]Suunto and Altimax are trademarks of Suunto Oy.

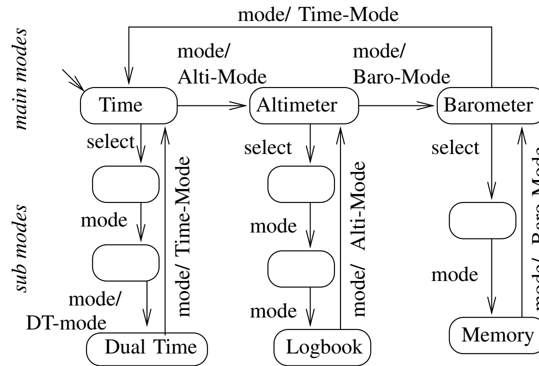


Figure 15: The Argos program for the Altimax watch.

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.

5.1.2 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.

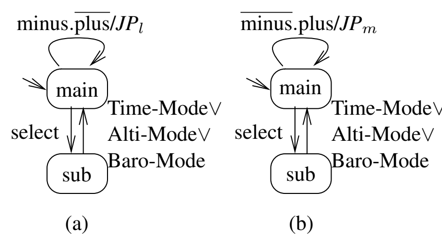


Figure 16: The pointcuts for the aspects.

These shortcuts can be implemented easily with Larissa aspects. Fig. 16 (a) shows the pointcut for the logbook aspect, and Fig. 16 (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, and JP_l or JP_m are no longer emitted. Furthermore, we use toInit advice with traces leading 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.

5.2 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 submode. 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: $\texttt{watch} \triangleleft \texttt{logbook} \triangleleft \texttt{memory} \not\approx \texttt{watch} \triangleleft \texttt{memory} \triangleleft \texttt{logbook}$.

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: $P \triangleleft \mathcal{A}_1 \triangleleft \mathcal{A}_2 \not\approx P \triangleleft \mathcal{A}_2 \triangleleft \mathcal{A}_1$. 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 16 (Joint weaving of several aspects)**.** *Let $\mathcal{A}_1 \dots \mathcal{A}_n$ be some aspects, with $\mathcal{A}_i = (P_{JP_i}, adv_i)$, and P a program. We define the application of $\mathcal{A}_1 \dots \mathcal{A}_n$ on P as follows:*

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

Note that Definition 16 reuses the advice weaving operator defined in Definition 5, and indexes the join point signal used by each advice. Furthermore, the advice is woven in the reverse order, i.e. we first the advice from the last aspect in the aspect list, and the advice from the first aspect last. This way, aspects that are later in the list have higher priority: if a join point transition is claimed by several aspects, the one that is woven first replaces the join point transition with its advice transition, and removes the join point signals of the other aspects. To give priority to the aspects that are applied later is consistent with sequential weaving, where aspects that are applied later modify the aspects that have already been applied, but not the other way round.

Jointly weaving the logbook and the memory aspect leads to the intended behavior, i.e. both aspects can be activated only when the program is in a main mode. Furthermore, 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 16 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. Then, the shortcut aspects must be sequentially woven *after* this aspect, so that they can select the new main mode as join point.

Definition 16 does not solve all conflicts. Indeed, the $\mathcal{A}_i$ in $P \triangleleft (\mathcal{A}_1, \dots, \mathcal{A}_n)$ do not commute, in general, since the advice weaving is applied sequentially. We define aspect interference for the application of several aspects.

Definition 17 (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*

$$P \triangleleft (\mathcal{A}_1 \dots \mathcal{A}_i, \mathcal{A}_{i+1} \dots \mathcal{A}_n) \not\approx P \triangleleft (\mathcal{A}_1 \dots \mathcal{A}_{i+1}, \mathcal{A}_i \dots \mathcal{A}_n)$$

As an example for interfering aspects, assume that the condition of the join point transition of the pointcut of the logbook aspect (Fig. 16 (a)) is only `minus` and the condition of the join point transition of the pointcut of the logbook aspect (Fig. 16 (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 Fig. 16, so that neither aspect executes when both buttons are pressed.

5.3 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 16. We can prove non-interference of two given aspects either for any program, or for a given program. Following [8], we speak of *strong independence* in the first case, and of *weak independence* in the second.

We use the operator *jpTrans* 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 18. *Let* $A = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $JP \in \mathcal{O}$*. Then,*

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

The following theorem proves strong independence between two aspects.

Theorem 2 (Strong Independence)**.** *Let* $\mathcal{A}_1 \dots \mathcal{A}_n$ *be some aspects, with* $\mathcal{A}_i = (P_{JP_i}, adv_i)$*. Then, the following equation holds:*

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

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 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 2 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 3 (Weak Independence)**.** *Let* $\mathcal{A}_1 \dots \mathcal{A}_n$ *be some aspects, with* $\mathcal{A}_i = (P_{JP_i}, adv_i)$*, and* $P_{pc} = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n})$*. Then, the following equation holds:*

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

See appendix C for a proof. Theorem 3 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 2 since it also takes the program P into account. However, there are cases in which the condition of Theorem 2 is false (thus it yields no results), but Theorem 3 allows to prove non-interference, e.g. in the case of the gangway aspects from Section 4, which is discussed in Section 5.3.2.

Theorem 3 is a sufficient condition, but, as Theorem 2, 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 3 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 pointcuts do 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 $P_{JP_1} \| \ \dots \ \| P_{JP_n} = P_{\text{all}\, JP}$: the condition of Theorem 2 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 3. 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 $P \triangleleft (\mathcal{A}_1 \dots \mathcal{A}_i, \mathcal{A}_{i+1} \dots \mathcal{A}_n) \sim P \triangleleft (\mathcal{A}_1 \dots \mathcal{A}_{i+1}, \mathcal{A}_i \dots \mathcal{A}_n)$, 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 16. 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. Indeed, the advice of an aspect influences which transitions are selected by the pointcut of an aspect that is sequentially woven next.

5.3.1 Interference between the Shortcut Aspects

Fig. 17 (a) shows the product of the pointcuts of the logbook (Fig. 16 (a)) and the memory aspect (Fig. 16 (b)). There are no transitions that emit both JP_l and JP_m, thus, by applying Theorem 2, we know that the aspects do not interfere, independently of the program they are applied to.

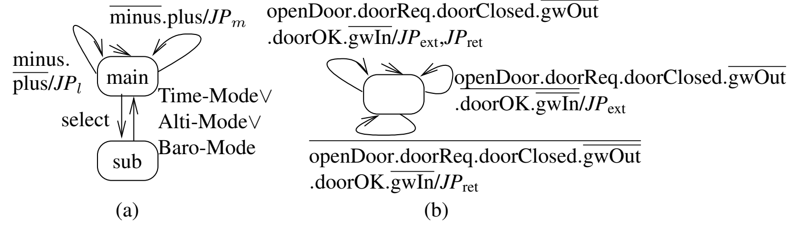


Figure 17: Interference between pointcuts.

Let us assume again that the condition of the join point transition of the pointcut of the logbook aspect (Fig. 16 (a)) is only `minus` and the condition of the join point transition of the pointcut of the logbook aspect (Fig. 16 (b)) is only `plus`. In this case, the state main in Fig. 17 (a) would have another loop transition, with label `minus.plus/`JP_l`,` JP_m. Thus, Theorem 2 not only states that the aspects potentially interfere, but it also states precisely 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 can solve the conflict.

5.3.2 Interference between the Gangway Aspects

As an example for weak interference, let us examine the gangway aspects from the Tram example in Section 4. Fig. 17 (b) shows the product of their pointcuts. There is a transition that has both JP_{ext} and JP_{ret} as outputs. Theorem 2 states that the aspects may interfere, but when applied to the tram controller from Fig. 13, they do not. This is because `doorOK` and `openDoor` are outputs of the controller, and are never emitted at the same time.

In this example, the use of Theorem 3 is thus needed to show that the aspects do not interfere when applied to the wristwatch controller. As expected, JP_{ext} and JP_{ret} are never emitted at the same time in P_{pc}, and Theorem 3 thus shows that the aspects do not interfere for this base program.

6 Related Work

Contracts and Aspects. Goldman and Katz [10] modularly verify aspect-oriented programs using a LTL tableau representation of programs and aspects. As opposed to ours, their system can verify AspectJ aspects, as tools like Bandera [7] can extract suitable input models from Java programs. It is, however, limited to so-called *weakly invasive* aspects, which only return to states already reachable in the base program.

Clifton and Leavens [5] noted before us that aspects invalidate the specification of modules, and propose that either an aspect should not modify a program's contract, or that modules should explicitly state which aspects may be applied to them.

Aspect Interference. Some authors discuss the advantages of sequential vs. joint weaving. Lopez-Herrejon and Batory [17] propose to use sequential weaving for incremental software development. Colyer and Clement [6, 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 [23] 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 [8] 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 [22] 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 [9] 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.

7 Conclusion

We presented two formal analysis tools for Larissa, which both exploit its semantic definition. The first combines Larissa with design-by-contract, and shows exactly how a Larissa aspect modifies the contract of a component to which it is applied. This allows us to calculate the effect of an aspect on a specification instead of only on a concrete program. This approach has several advantages. First, aspects can be checked against contracts even if the final implementation is not yet available during development. Furthermore, if the base program is changed, the woven program must not be re-verified, as long as the new base program still fulfills the contract. Finally, woven programs can be verified modularly, which may allow to verify larger programs.

The second approach is an analysis for aspect interference in Larissa. 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 studied, the conflicts were solved by simple modifications of the pointcuts.

These analysis are only possible because Argos and Larissa are very simple languages with clean and simple semantics. They thus illustrate the advantage of using a programming language with simple semantics. Because of this simplicity, both approaches seem quite precise. Indeed, We believe that the contract weaving is exact in that it gives no more possible behaviors for the woven program than necessary. I.e., for

a contract C and a trace $t \in \mathit{Traces}(C \triangleleft asp)$, there exists a program P s.t. $P \models C$ and $t \in \mathit{Traces}(P \triangleleft asp)$. This remains however to be proven.

The interference analysis for Larissa seems also 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 [8], assuming that a shared join point leads to a conflict does not introduce spurious conflicts.

There are some interesting points for further work. In the context of contract weaving, an interesting question is if we can derive contracts the other way round. Given a contract C and an aspect asp, can we automatically derive a contract C' such that $C' \triangleleft asp \models C$? Finally, both approaches work only because we have restricted Argos and Larissa to Boolean signals. It would be interesting to see if they can be extended to programs with variables.

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*, 63(3):297–320, 2006. 1, 2, 2.3

[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. 5.1, 5.1.1

[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. 6

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19(2):87–152, 1992. 1, 2.2

[5] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-15, Iowa State University, Department of Computer Science, Dec. 2003. 6

[6] A. Colyer and A. Clement. Large-scale AOSD for middleware. In K. Lieberherr, editor, *AOSD-2004*, pages 56–65, Mar. 2004. 6

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, June 2000. 6

[8] 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. 5.3, 6, 7

[9] 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. 6

[10] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect-Oriented Languages (FOAL)*, Mar. 2006. 6

[11] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993. 1

[12] 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. 1, 2.3.1

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–353, 2001. 1

[14] L. Lamport. Proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, SE-3(2):125–143, 1977. 2.3.1

[15] Compiler for Larissa. `http://www-verimag.imag.fr/~stauch/ArgosCompiler/`. 4

[16] Argos source code for the tram example. `http://www-verimag.imag.fr/~stauch/ArgosCompiler/contracts.html`. 4

[17] 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. 6

[18] The Lustre tutorial. `http://www-verimag.imag.fr/~raymond/edu/tp.ps.gz`. 4

[19] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, Aug. 2004. 1

[20] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001. 1, 2, 5.1.1

[21] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992. 1

[22] 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. 6

[23] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, Sept. 2003. 6

[24] D. Stauch. *Larissa, an Aspect-Oriented Language for Reactive Systems*. PhD thesis, Institut National Polytechnique de Grenoble, Nov. 2007. 2.3.2, 3.3

[25] D. Stauch. Modifying contracts with Larissa aspects. In *Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P)*, Braga, Portugal, Mar. 2007. ENTCS. To appear. 1

[26] D. Stauch, K. Altisen, and F. Maraninchi. Interference of Larissa aspects. In *Foundations of Aspect-Oriented Languages (FOAL)*. published as Iowa State University Technical Report #06-01, Mar. 2006. 1

A Proof for Theorem 1

Definitions. We first introduce a number of definitions.

$P(p) \models (A(a), G(g))$ means that program P fulfills contract (A, G) where the initial states of P, A and G have been set to p, a and g respectively.

Furthermore, we introduce the following notations for terms from the theorem. Let

$$A' \triangleleft asp = \mathit{OBS}_A(\mathit{ND}_A(A) \triangleleft asp), \qquad A \triangleleft asp = Det_A(A' \triangleleft asp),$$
$$G' \triangleleft asp = \mathit{OBS}_G(\mathit{ND}_G(G) \triangleleft asp), \text{ and} \qquad G \triangleleft asp = Det_G(G' \triangleleft asp) .$$

We now define the structure of some of these terms. Let

$$\begin{aligned}
P &= (\mathcal{Q}_P, q_{P0}, \mathcal{I}, \mathcal{O}, \mathcal{T}_P),\\
asp &= (\text{PC}, < O_{adv}, \text{toInit}, \sigma >),\\
\text{PC} &= (\mathcal{Q}_{\text{PC}}, q_{\text{PC}0}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{\text{PC}}),\\
A &= (\mathcal{Q}_A \cup \{\text{Error}\}, q_{A0}, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, \mathcal{T}_A),\\
G &= (\mathcal{Q}_G \cup \{\text{Error}\}, q_{G0}, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, \mathcal{T}_G),\\
P \triangleleft asp &= (\mathcal{Q}_P \times \mathcal{Q}_{\text{PC}}, (q_{P0}, q_{\text{PC}0}), \mathcal{I}, \mathcal{O}, \mathcal{T}_{P\triangleleft}),\\
A' \triangleleft asp &= ((\mathcal{Q}_A \times \mathcal{Q}_{\text{PC}}) \cup \{\text{Error}\}, (q_{A0}, q_{\text{PC}0}), \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, \mathcal{T}_{A\triangleleft}), \text{ and}\\
G' \triangleleft asp &= ((\mathcal{Q}_G \times \mathcal{Q}_{\text{PC}}) \cup \{\text{Error}\}, (q_{G0}, q_{\text{PC}0}), \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, \mathcal{T}_{G\triangleleft})\ .
\end{aligned}$$

We prove the theorem by induction over a trace of $P \triangleleft asp$. Let $(it, ot) \in \mathit{Traces}(P \triangleleft asp)$. We show that the following induction hypothesis holds for any n.

Induction hypothesis. The induction hypothesis states that the states reached by executing (it, ot) on $P \triangleleft asp$, $A' \triangleleft asp$, and $G' \triangleleft asp$ formed a valid contract in P, A, and G, i.e. before the aspect was applied, provided (it, ot) is accepted by $A \triangleleft asp$. Formally, we write it as follows:

$$\begin{aligned}
&O_step_{A\triangleleft asp}((s_{A0}, s_{\text{PC}0}), it.ot, n) = \emptyset \wedge\\
&(p_n, pc_n) = S_step_{P\triangleleft asp}((s_{P0}, s_{\text{PC}0}), it, n)\\
&\quad \Rightarrow \exists (a_n, pc_n) \in S_step_{A'\triangleleft asp}((s_{A0}, s_{\text{PC}0}), it.ot, n)\ .\\
&\quad\ \ \exists (g_n, pc_n) \in S_step_{G'\triangleleft asp}((s_{G0}, s_{\text{PC}0}), it.ot, n)\ .\\
&\quad\quad P(p_n) \models (A(a_n), G(g_n)) \wedge g_n \neq \text{Error}
\end{aligned}$$

(p_n, pc_n), (a_n, pc_n) and (g_n, pc_n) are the states reached when executing (it, ot) for n steps on $P \triangleleft asp$, $A' \triangleleft asp$ and $G' \triangleleft asp$ respectively.

Base case. $n = 0$. $P \models (A, G)$ holds as it is the assumption of the implication in the theorem. If the initial state of G is the Error State, either A (and $A \triangleleft asp$) do not accept any trace, or no P exists, and in both cases we are done.

Induction step. From $n - 1$ to n.

If $O_step_{A\triangleleft asp}(it.ot, n) = \{\{\texttt{err}\}\}$, we are done. Otherwise, from $O_step_{A\triangleleft asp}(it.ot, n) = \{\emptyset\}$ follows $O_step_{A'\triangleleft asp}(it.ot, n) = \{\emptyset\}$, because Definition 14, which defines the determinization of A', gives precedence to error transitions. We distinguish two cases:

- First case: $\{JP\} \notin O_step_{\text{PC}}(it.ot, n)$, we are not in a join point.

  Because of $P(p_{n-1}) \models (A(a_{n-1}), G(g_{n-1}))$, there is a transition $t_p = (p_{n-1}, it(n), ot(n), p_n)$ in $\mathcal{T}_P$, a transition $t_a = (a_{n-1}, it(n) \wedge ot(n), \emptyset, a_n)$ in $\mathcal{T}_A$, and a transition $t_g = (g_{n-1}, it(n) \wedge ot(n), \emptyset, g_n)$ in $\mathcal{T}_G$, such that $P(p_n) \models (A(a_n), G(g_n))$. t_p, t_a and t_g are not modified by the weaving, thus there is a transition $((p_{n-1}, pc_{n-1}), it(n), ot(n), (p_n, pc_n))$ in $\mathcal{T}_{P\triangleleft}$, a transition $((a_{n-1}, pc_{n-1}), it(n) \wedge ot(n), \emptyset, (a_n, pc_n))$ in $\mathcal{T}_{A\triangleleft}$, and a transition $((g_{n-1}, pc_{n-1}), it(n) \wedge ot(n), \emptyset, (g_n, pc_n))$ in $\mathcal{T}_{G\triangleleft}$ with $(g_n, pc_n) \neq$ Error.

- Second case: $\{JP\} \in O_step_{\text{PC}}(it.ot, n)$, we are in a join point.

  Let $p_\sigma = S_step_P(s_{P0}, \sigma, l_\sigma)$ be the state in the P reached after executing σ, and let ς be a trace of length l_σ such that $\forall i \leq l_\sigma\ .\ \varsigma(i) = O_step_P(s_{P0}, \sigma, i)$. Then, let $S_step_{\text{PC}}(s_{\text{PC}0}, \sigma.\varsigma, l_\sigma) = \text{pc}_\sigma$ be the state of the pointcut reached after executing σ. Then, we also have $S_step_{P\triangleleft asp}((s_{P0}, s_{\text{PC}0}), it, n) = (p_\sigma, \text{pc}_\sigma)$.

  All join point transitions in $G' \triangleleft asp$ (resp. $A' \triangleleft asp$) are replaced by transitions to all possible target states, thus there is a transition $t_{g'\triangleleft} \in \mathcal{T}_{G'\triangleleft}$ (resp. $t_{a'\triangleleft} \in \mathcal{T}_{A'\triangleleft}$) to a target state $(g_\sigma, \text{pc}_\sigma)$ (resp.

$(a_\sigma, \mathrm{pc}_\sigma)$) such that $S_step_G(s_{G0}, \sigma.\varsigma, l_\sigma) = g_\sigma$ (resp. $S_step_A(s_{A0}, \sigma.\varsigma, l_\sigma) = a_\sigma$). Because p_σ, a_σ and g_σ can be reached with the same trace (σ, ς) (resp. $(\sigma.\varsigma, \diamond)$ for a_σ and g_σ) from the initial state, $P(p_\sigma) \models (A(a_\sigma), G(g_\sigma))$ follows from $P \models (A, G)$.

Furthermore, $ot(n) = \ell_{0_{adv}} \wedge \ell_{\overline{\mathcal{O} \setminus 0_{adv}}}$, and we have $t_{a'\triangleleft} = ((a_{n-1}, pc_{n-1}), it(n) \wedge ot(n), \emptyset, (a_\sigma, pc_\sigma))$, and $t_{g'\triangleleft} = ((g_{n-1}, pc_{n-1}), it(n) \wedge ot(n), \emptyset, (g_\sigma, pc_\sigma))$, and thus $(a_\sigma, pc_\sigma) = S_step_{A'\triangleleft asp}((s_{A0}, s_{\mathrm{PC}0}), it.ot, n)$ and $(g_\sigma, pc_\sigma) = S_step_{G'\triangleleft asp}((s_{G0}, s_{\mathrm{PC}0}), it.ot, n)$. Furthermore, we have $(g_\sigma, pc_\sigma) \neq$ Error, as otherwise $a_\sigma =$ Error (impossible because of $O_step_{A'\triangleleft asp}((s_{A0}, s_{\mathrm{PC}0}), it.ot, n) = \emptyset$), or $(it, ot) \notin \mathit{Traces}(P)$, by the definition of $P \models (A, G)$.

It follows from the induction hypothesis that

$$(it.ot, \diamond) \in \mathit{Traces}(A \triangleleft\, asp) \wedge (it, ot) \in \mathit{Traces}(P \triangleleft\, asp) \Rightarrow (it.ot, \diamond) \in \mathit{Traces}(G' \triangleleft\, asp)$$

and we have $(it.ot, \diamond) \in \mathit{Traces}(G' \triangleleft\, asp) \Rightarrow (it.ot, \diamond) \in \mathit{Traces}(G \triangleleft\, asp)$ by Definition 15. Thus, the theorem follows from the induction hypothesis. □

B Proof for Theorem 2

Theorem 2 and Theorem 3 are both implications with the same consequent.

We show that the antecedent of the implication in Theorem 3,

$$\begin{aligned} &\mathit{jpTrans}(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_i) \cap \\ &\mathit{jpTrans}(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_{i+1}) = \emptyset, \end{aligned}$$

follows from the antecedent of the implication in Theorem 2,

$$\begin{aligned} &\mathit{jpTrans}(P_{JP_i} \| P_{JP_{i+1}}, JP_i) \\ &\cap \mathit{jpTrans}(P_{JP_i} \| P_{JP_{i+1}}, JP_{i+1}) = \emptyset : \end{aligned}$$

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 $P_{JP_i} \| P_{JP_{i+1}}$.

Thus, because of the transitivity of the implication, Theorem 2 is a consequence of Theorem 3. □

C Proof for Theorem 3

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

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

$$\begin{aligned} &\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \\ &\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O\big) \implies \\ &\qquad (s, \ell, O_{adv_{i+1}}, S_step_{P'}(s_{\mathrm{init}}, \sigma_{i+1}, l_{\sigma_{i+1}})) \in \mathcal{T}' \end{aligned}$$

and $P_{i+1} \triangleleft_{JP_i} adv_i$ yields an automaton $P_i = (\mathcal{Q}, s_{\mathrm{init}}, \mathcal{I}, \mathcal{O} \cup O_{adv_{i+1}} \cup O_{adv_i}, \mathcal{T}'')$, where $\mathcal{T}''$ is defined as follows:

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \tag{7}$$

$$\begin{aligned} &\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \notin O\big) \implies \\ &\qquad (s, \ell, O_{adv_{i+1}}, S_step_{P'}(s_{\mathrm{init}}, \sigma_{i+1}, l_{\sigma_{i+1}})) \in \mathcal{T}' \end{aligned} \tag{8}$$

$$\begin{aligned} &\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \in O\big) \implies \\ &\qquad (s, \ell, O_{adv_i}, S_step_{P'}(s_{\mathrm{init}}, \sigma_i, l_{\sigma_i})) \in \mathcal{T}' \end{aligned} \tag{9}$$

$$\begin{aligned} &\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies \\ &\qquad (s, \ell, O_{adv_{i+1}}, S_step_{P'}(s_{\mathrm{init}}, \sigma_{i+1}, l_{\sigma_{i+1}})) \in \mathcal{T}' \end{aligned} \tag{10}$$

If we calculate $P_{i+2} \lhd_{JP_i} adv_i \lhd_{JP_{i+1}} adv_{i+1}$, we obtain the same automaton, except for transitions (10), which are defined by

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

Transitions (10) are exactly the join point transitions that are in $jpTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_i) \cap jpTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}), JP_{i+1})$. 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 (10) has been added by the weaving of $\lhd_{JP_n} adv_n \ldots \lhd_{JP_{i+2}} adv_{i+2}$.

Thus, we have $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \lhd_{JP_n} adv_n \ldots \lhd_{JP_{i+2}} adv_{i+2} \lhd_{JP_{i+1}} adv_{i+1} \lhd_{JP_i} adv_i = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \lhd_{JP_n} adv_n \ldots \lhd_{JP_{i+2}} adv_{i+2} \lhd_{JP_i} adv_i \lhd_{JP_{i+1}} adv_{i+1}$. Weaving $\lhd_{JP_{i-1}} adv_{i-1} \ldots \lhd_{JP_1} adv_1$ trivially yields the same result. $\square$