

Modifying Contracts with Larissa Aspects

David Stauch

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

January 22, 2007

Abstract:

This paper combines two successful techniques from software engineering, aspect-oriented programming and design-by-contract, and applies them in the context of reactive systems. For the aspect language Larissa and contracts expressed with synchronous observers, 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'*. We validate the approach on a medium-sized example. A shorter version of report has been submitted to the SLA++P '07 workshop.

1 Introduction

1.1 Synchronous Languages and Aspect-Oriented Programming

Aspect-oriented programming (AOP) offers facilities to a base language which aim at encapsulating *cross-cutting 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 program with an aspect weaver.

All the aspect extensions of existing languages (like AspectJ [7]) 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 the methods of the class *X*, all the methods whose name contains *visit*, etc.). The *advice* specifies what has to be done at each join point (execute a piece of code before the normal code of the method, for instance).

Most existing aspect languages cannot be used in the context of reactive systems, because they lack the semantic properties needed for formal verification, and the programming languages used for reactive systems are often different from general-purpose programming languages. Therefore, we developed the aspect language Larissa [1] as an extension to the synchronous programming language Argos. Argos is a hierarchical automata language, based on Mealy machines. It seems a good candidate as a base language, as it is the simplest language with the parallel structure which we want to crosscut, and which is typical for synchronous languages. Larissa has strong semantic properties, like the preservation of equivalence between programs. The approach presented in this paper strongly depends on these properties.

1.2 Synchronous Languages and Design-by-Contract

Design-by-Contract [14] is a design principle, originally introduced for object-oriented systems, where a method is specified by a contract. A contract is a specification in form of an implication between an assumption clause and a guarantee clause. A method fulfills its contract if after its execution, the guarantee holds if the assumption was true when the program was called.

Contracts have been adapted to reactive systems by [12]. Reactive systems constantly receive inputs from their environment, and emit outputs to it. Therefore, it seems natural to let assumptions restrict the inputs, and let guarantees ensure properties on the outputs. Additionally, what a program is allowed to do often depends to a large extent on previous occurrences of signals. A convenient way to express such temporal properties over input and output traces are observers. An observer [6] is a program that observes the inputs and the outputs of the program, without modifying its behavior, and computes a safety property (in the sense of safety/liveness properties as defined in [8]). Observers have a single output `err`, which is emitted to show that a trace is not accepted. They can be expressed in the same language as the program.

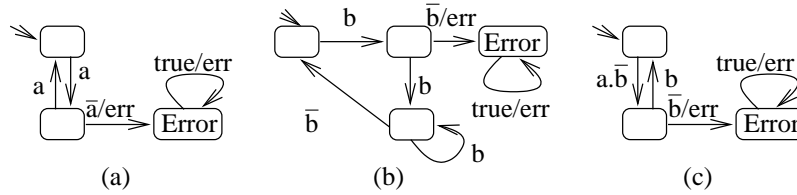


Figure 1: The contract for the MFF. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by “triggering condition / outputs emitted”, e.g. a transition labelled by “a/b” is triggered when a is true and emits b. Negation is expressed with an overbar, and conjunction with a dot.

```

1  class c{
2    /* @assume i < 10 */
3    /* @guarantee \result < 10 */
4    int m(int i){ ... }
5  }
6
7  pointcut pcm(int i) : call(int c.m(int)) && args(i);
8
9  int around(int i) : pcm(i){
10   return 1 + proceed(i+1);
11 }

```

Figure 2: Example of a contract in presence of an AspectJ aspect.

As an example, consider the following contract for a mono-stable flip-flop (MFF) with one input a and one output b . The contract is composed of an assumption, shown in Figure 1(a), which states that a 's always occur in pairs, and a guarantee consisting of two automata, shown in Figures 1(b) and (c), which are composed in parallel. The automaton in Figure 1(b) guarantees that a single b is never emitted, and the automaton in Figure 1(c) guarantees that when a occurs while no b is emitted, b is emitted in the next instant.

1.3 Combining Contracts and Aspects

AOP and design-by-contract can hardly be used concurrently. Obviously, the contract of a program is invalidated when an aspect is applied to it. Consider the AspectJ example in Figure 2. The pointcut (line 7) intercepts calls to method m (line 4), and the around advice (lines 9–11) modifies the intercepted calls by adding 1 to the argument, then calling m through the proceed statement, and adding 1 to the result. This modifies both the initial assumption (line 2) and guarantee (line 3) of m . However, we can give a new contract for m in this case. To ensure that m is called according to its initial specification, the assumption must be changed to $i < 9$. On the other hand, the value returned by m may be higher than specified by the original guarantee in the presence of the aspect: we can only guarantee that $\text{\result} < 11$, provided m does not call itself recursively.

Deriving such new contracts appears to be an interesting approach to combine AOP and contracts. However, this seems very difficult for contracts for Java programs and AspectJ, and it is not clear if meaningful contracts could be derived. In this paper, we present a way to derive new contracts for Argos programs and Larissa aspects. The idea is to apply an aspect asp to a contract C and obtain a new contract C' , such that if P fulfills C , then $P \triangleleft asp$ fulfills C' .

The remainder of the paper is structured as follows: Section 2 defines Argos and Larissa; Section 3 describes how to derive a new contract from a contract and an aspect; Section 4 validates the approach on a larger example; Section 5 describes related work; and Section 6 concludes.

2 Argos and Larissa

This section presents a restriction of the Argos language [13], 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 sets of Boolean input and output variables representing signals from and to the environment. An input trace, it , is a function: $it : N \rightarrow [\mathcal{I} \rightarrow \{\text{true}, \text{false}\}]$. An output trace, ot , is a function: $ot : N \rightarrow [\mathcal{O} \rightarrow \{\text{true}, \text{false}\}]$. 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 $\forall (it, ot), (it', ot') \in S. (it = it') \implies (ot = ot')$.

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

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

2.2 Argos

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

The synchronous product allows to put automata in parallel which synchronize on their common inputs. The encapsulation is the operator that expresses the communication between automata with the synchronous broadcast: if two automata are put in parallel, they can communicate via a signal s . This signal is an input of the first automaton and an output of the second. The encapsulation operator computes this communication and then hides the signal s . The semantics of an automaton is defined by a set of traces, and the semantics of the operators is given by translating expressions into flat automata.

Definition 2 (Automaton). An automaton \mathcal{A} is a tuple $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where \mathcal{Q} is the set of states, $s_{init} \in \mathcal{Q}$ is the initial state, \mathcal{I} and \mathcal{O} are the sets of Boolean input and output variables respectively, $\mathcal{T} \subseteq \mathcal{Q} \times Bool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $Bool(\mathcal{I})$ 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, $\ell \in Bool(\mathcal{I})$ 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 $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is given in terms of a set of pairs of i/o-traces. This set is built using the following functions:

$$S_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times N \rightarrow \mathcal{Q}$$

$$O_step_{\mathcal{A}} : \mathcal{Q} \times InputTraces \times N \setminus \{0\} \rightarrow 2^{\mathcal{O}}$$

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

$$n = 0 : S_step_{\mathcal{A}}(s, it, n) = s$$

$$n > 0 : S_step_{\mathcal{A}}(s, it, n) = s' \quad O_step_{\mathcal{A}}(s, it, n) = O$$

where $\exists (S_step_{\mathcal{A}}(s, it, n-1), \ell, O, s') \in \mathcal{T} \wedge \ell$ has value true for $it(n-1)$.

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

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

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

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

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

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

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. $\ell^- = \{x \in \mathcal{I} \mid (\bar{x} \wedge \ell) = \ell\}$).

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

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

In general, the encapsulation operation does not preserve determinism nor completeness. This is related to the so-called ‘‘causality’’ problem intrinsic to synchronous languages (see, for instance [2]).

2.3 Contracts for Argos

An observer is an automaton which specifies a class of programmes fulfilling a certain safety property. It is formally defined as follows.

Definition 5 (Observer). An observer is an automaton $(\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\text{err}\}, \mathcal{T})$ which observes an automaton that has 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 $P \models obs$) iff $P \parallel obs \setminus \mathcal{O}$ 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 6 is an auxiliary definition, used to formally define contracts in Definition 7. ϵ denotes the empty trace.

Definition 6 (Trace Combination). Let $it : N \rightarrow [\mathcal{I} \rightarrow \{\text{true}, \text{false}\}]$ and $ot : N \rightarrow [\mathcal{O} \rightarrow \{\text{true}, \text{false}\}]$ be traces, with $\mathcal{I} \cap \mathcal{O} = \emptyset$. Then, $it.ot : N \rightarrow [\mathcal{I} \cup \mathcal{O} \rightarrow \{\text{true}, \text{false}\}]$ 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 7 (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

$$(it.ot, \epsilon) \in Traces(A) \Rightarrow ((it, ot) \in Traces(P) \Rightarrow (it.ot, \epsilon) \in Traces(G)) .$$

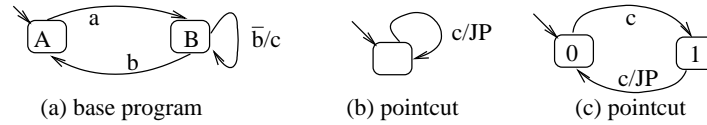


Figure 3: Example pointcut.

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 7 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 .

2.4 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 precisely to specify such cross-cutting modifications of a program, 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.

2.4.1 Join Point Selection.

To preserve semantical equivalence, pointcuts in Larissa are not expressed in terms of the internal structure of the base program (as for instance state names), but refer to the observable behavior of the program only, i.e., its inputs and outputs.

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 \mathcal{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, \mathcal{JP} is emitted when T is taken.

Technically, we perform a parallel product between the program and the pointcut and select those transitions in the product which emit \mathcal{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 $\{o_1, \dots, o_n\}$, is given by:

$$\mathcal{P}(P, PC) = (P' \parallel PC' \parallel dupl_{o_1} \parallel \dots \parallel dupl_{o_n}) \setminus \{o'_1, \dots, o'_n\}$$

The join point transitions are those transitions of $\mathcal{P}(P, PC)$ that emit \mathcal{JP} .

Figure 3 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.

2.4.2 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*

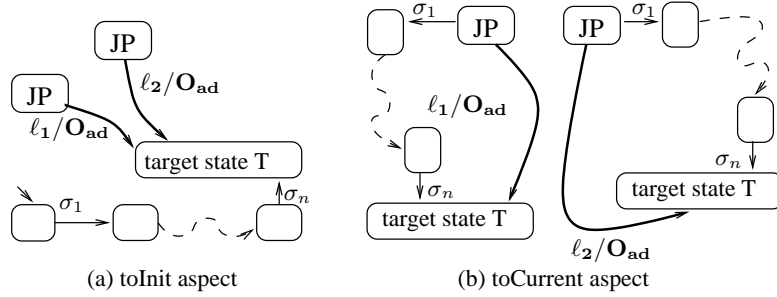


Figure 4: Schematic toInit and toCurrent aspects. (Advice transitions are in bold.)

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 $\triangleleft adv$ weaves a piece of advice adv in a program. Definition 10 in the following section gives a formal definition for toInit advice. The remainder of this section describes the different kinds of advice informally.

Advice Transition. 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_{ad} . These are the same for every advice transition, and are thus specified in the aspect. Advice transitions are illustrated in Figure 4.

Advice Program. 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 *terminates*. 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 the *advice outputs* O_{ad} ; 2) connect the transitions that went to the final state F in A_{ins} to T . Advice programs are illustrated in Figure 5.

2.4.3 Fully Specifying an Aspect.

An aspect is given by the specification of its pointcut and its advice: $asp = (PC, adv)$, where PC is the pointcut and adv is the advice. adv is a tuple which contains 1) the advice outputs O_{ad} ; 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 $\langle O_{ad}, type, \sigma \rangle$, or, with an advice program, a tuple $\langle O_{ad}, type, \sigma, P_{adv} \rangle$, with $type \in \{toCurrent, toInit\}$. An aspect is woven into a program by first determining the join point transitions and then weaving the advice.

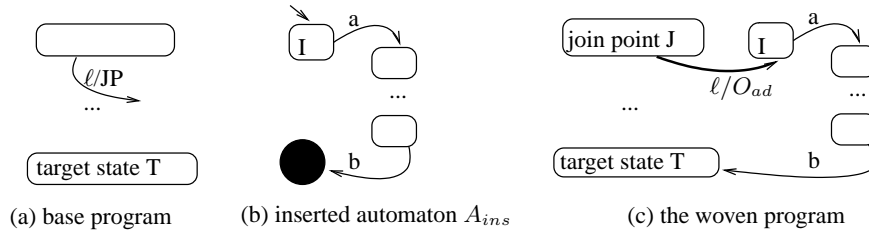


Figure 5: Inserting an advice automaton

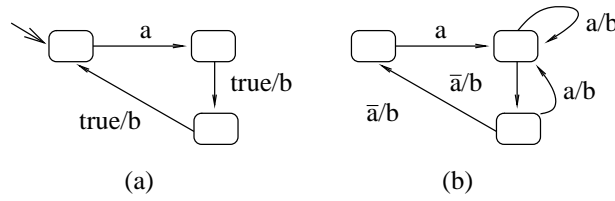


Figure 6: A possible implementation of the MFF (a), with the retriggerable aspect applied to it (b).

Definition 8 (Aspect weaving). Let P be a program and $asp = (PC, adv)$ an aspect for P . The weaving of asp on P is defined by

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

Example.

Consider the MFF example from Section 1.2. We now want to make the MFF re-triggerable, meaning that if an a is emitted during several following instants, the MFF continues emitting b . We do this by applying the aspect $ret = (PC, \langle b, toInit, (a) \rangle)$ 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. Figure 6(b) shows the result of applying ret to the sample implementation of the MFF in Figure 6(a).

3 Weaving Aspects in Contracts

We want to apply an aspect asp not to a specific program, 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 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 the aspect weaving, all of these steps are different for the assumption and the guarantee, as far as the transitions to the Error state are concerned. This is because the assumption and the guarantee have different functions in a contract: the assumption defines which part of the program is defined by the contract, and the guarantee gives properties that are always true for this part.

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.

Indeed, a contract (A, G) can be rewritten as $(\text{true}, A \Rightarrow G)$. Thus, the assumption can be considered as a negated guarantee.

3.0.4 Formal Definitions.

Definition 9 defines the transformation of an observer into a NDA through two functions, one for guarantee observers and one for assumption observers.

Definition 9 (Observer to NDA transformation). Let $obs = (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I} \cup \mathcal{O}, \{err\}, T)$ 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 $(s, \ell_{\mathcal{I}} \wedge \ell_{\mathcal{O}}, o, s') \in T \Rightarrow (s, \ell_{\mathcal{I}}, \ell_{\mathcal{O}}^+ \cup o, s') \in T_{ND_A}$.

Note that the transitions in *obs* which emit *err* 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. In the assumption, on the other hand, they correspond to inputs from the environment that are not treated by the program. 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.

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.

Definition 10 (toInit weaving for NDA). Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, T)$ be an automaton and $adv = (O_{adv}, toInit, \sigma)$ a piece of *toInit* advice, with $\sigma : [0, \dots, \ell_{\sigma}] \rightarrow [\mathcal{I} \rightarrow \{\text{true}, \text{false}\}]$ a finite input trace of length $\ell_{\sigma} + 1$. Let $TARG = \{s \mid s = S_step_{\mathcal{A}}(s_{init}, \sigma, \ell_{\sigma})\}$ be the set of all states reachable with σ . The advice weaving operator \triangleleft , weaves *adv* into \mathcal{A} and returns the automaton $\mathcal{A} \triangleleft adv = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O} \cup O_{adv}, T')$, where T' is defined as follows:

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

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

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

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

$$(s, \ell, o, s') \in T \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O} \setminus o}}, \emptyset, s') \in T' \quad (3)$$

$$(s, \ell, \emptyset, s') \notin T' \wedge s \in \mathcal{Q} \wedge \ell \text{ is a complete monomial over } \mathcal{I} \cup \mathcal{O} \Rightarrow (s, \ell, \{err\}, Error) \in T'' \quad (4)$$

where $\ell_o = \bigwedge_{o \in \mathcal{O}} o$ and $\ell_{\bar{o}} = \bigwedge_{o \in \mathcal{O}} \bar{o}$ for a set O of variables.

Definition 12 (NDA to assumption transformation). Let $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O} \cup \{err\}, T)$ be a NDA. $OBS_A(nd) = (\mathcal{Q}, q_0, \mathcal{I} \cup \mathcal{O}, \{err\}, T')$ defines an observer, where T' is defined by

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

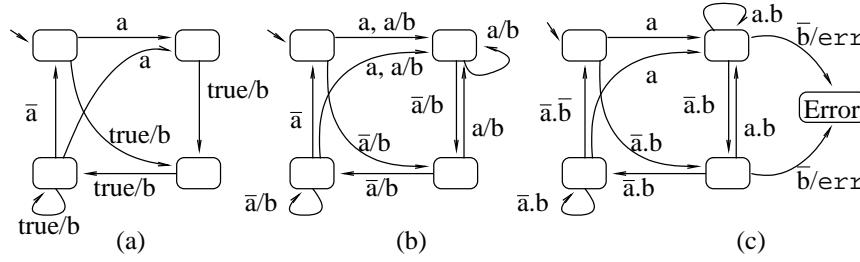


Figure 7: a: $ND_G(\text{gMFF})$, b: $ND_G(\text{gMFF}) \triangleleft \text{ret}$, c: $OBS_G(ND_G(\text{gMFF}) \triangleleft \text{ret})$.

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.

Definition 13 (Assumption Determinization). Let M be a NDA with outputs $\{\text{err}\}$. $Det_A(M)$ is a deterministic automaton such that

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

Definition 14 (Guarantee Determinization). Let M be a NDA with outputs $\{\text{err}\}$. $Det_G(M)$ is a deterministic automaton such that

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

We can now state the following theorem.

Theorem 1.

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

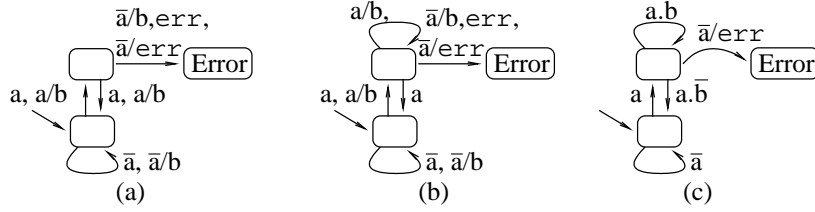
Proof. See appendix A.

3.0.5 Example.

We apply the re-triggerable aspect `ret` from Section 2.4.3 to the contract of the MFF using the method described above and obtain the contract of `MFF<ret`. Figure 7(a) shows $ND_G(\text{gMFF})$, the NDA equivalent to `gMFF`. Figure 7(b) shows $ND_G(\text{gMFF}) \triangleleft \text{ret}$, the same automaton with the aspect woven into it. Figure 7(c) shows this automaton, transformed back in an observer. It is already deterministic, thus there is no need to determinize it. Figure 8 describes the same steps for the assumption.

4 Example: The Tramway Door Controller

We implement and verify a larger example, taken from the Lustre tutorial [11], 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 gateway, which can be extended to allow passengers in wheelchairs enter and leave the tram.

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

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		
deplmm	the tramway wants to leave the station		

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

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. The in- and outputs of the controller and those which are added by the gangway are given in Figure 9. The controller uses additional inputs, called Helper Signals, which are shown in Figure 9 and are calculated from the original inputs by the program in Figure 10.

4.0.6 The Gangway Aspects.

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

The pointcut PC_{ext} of the extension aspect selects all transitions where $\text{openDoor.doorReq.doorClosed.gwOut}$ is true, and the pointcut PC_{ret} of the retraction aspect selects all transitions where doorOK.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 Figure 11. The extension aspect is specified by $(PC_{\text{ext}}, \langle \{ \}, \text{toCurrent}, (), I_{\text{ext}} \rangle)$, and the retraction aspect by $(PC_{\text{ret}}, \langle \{\text{retractGW}\}, \text{toCurrent}, (), I_{\text{ret}} \rangle)$.

4.0.7 Verifying Safety Properties.

The tramway system must fulfill several safety requirements concerning the doors, namely:

- the door must be closed while the tram is out of station,
- the gangway must be fully retracted while the tram is out of station, and
- The gangway must not be moved when the door is not closed.

We want to verify that the controller guarantees these requirements if it is put in an appropriate environment. Therefore, we develop a model that describes the possible behavior of the physical environment of the controller, composed of the door, the gangway, and the tramway. These models are expressed as Argos observers. The models for the tramway and the door are shown in Figure 12 and Figure 13 respectively.

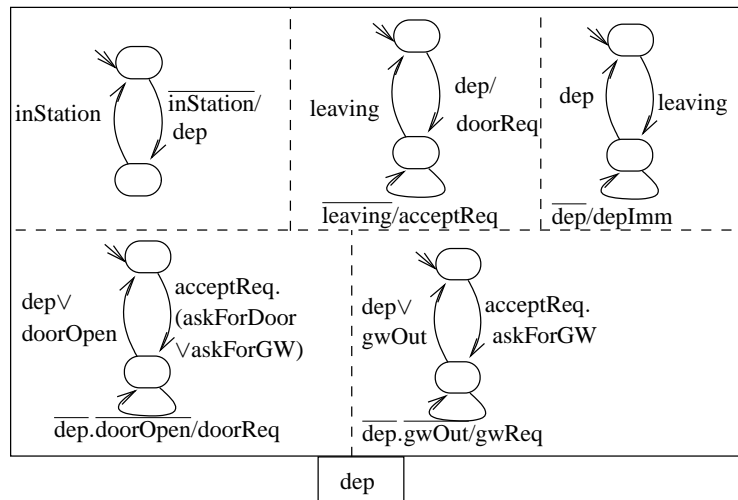


Figure 10: The automaton producing the helper signals.

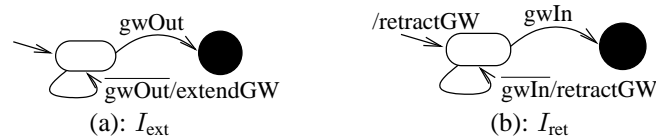


Figure 11: Inserted automata for the extension (a) and the retraction (b) aspect.

The model for the gangway is the same as for the door, where door-related signals are replaced by their gangway-related counterparts.

The actual door controller may be a large program which is difficult to verify, therefore we abstract it with a contract. The assumption of the contract is the model of the door shown in Figure 13, and the guarantee is shown in Figure 14. To prove that a controller without gangway satisfies the safety requirements, we prove that a) the controller satisfies the contract, and b) the contract never violates a safety requirement in the environment.

To verify modularly that the program with the gangway does not violate the safety requirements, we weave the aspects into the assumption and the guarantee, and check then that a) the environment satisfies the new assumption, and b) the new guarantee satisfies the safety requirements in the environment.

We measured the time necessary to verify the safety properties using our implementation [9]. The source code of the door controller example is available at [10]. Building the sample controller shown in Figure 15 with the gangway aspects and verifying it against the environment takes 11.0 seconds. Weaving the aspects into the guarantee of the controller contract and verifying against the environment takes 3.7 seconds, verifying that the sample controller verifies the contract and verifying that the environment fulfills the assumptions with the aspects takes < 0.5 seconds. Thus, using this modular approach to verify the safety properties of the controller is significantly faster than verifying the complete program. This indicates

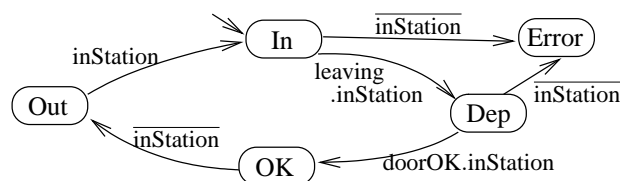


Figure 12: Model of the tramway.

6 Conclusion

We proposed a way to show 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. Furthermore, we used this approach to modularly verify an example.

We believe that the approach 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 $t \in \text{Traces}(C \triangleleft \text{asp})$, there exists a program P s.t. $P \models C$ and $t \in \text{Traces}(P \triangleleft \text{asp})$. This remains however to be proven. A more interesting direction for future work would be to derive contracts the other way round. Given a contract C and an aspect asp , can we automatically derive a contract C' such that $C' \triangleleft \text{asp} \models C$? Finally, the proposed approach works only because we have restricted Argos and Larissa to Boolean signals. It would be interesting to see if this approach can be extended to programs with valued signals or 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. 1.1, 2, 2.4
- [2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Programming*, 19(2):87–152, 1992. 2.2
- [3] 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. 5
- [4] 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. 5
- [5] M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect-Oriented Languages (FOAL)*, Mar. 2006. 5
- [6] 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
- [7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–353, 2001. 1.1
- [8] L. Lamport. Proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, SE-3(2):125–143, 1977. 1.2
- [9] Compiler for Larissa. <http://www-verimag.imag.fr/~stauch/ArgosCompiler/>. 4.0.7
- [10] Argos source code for the tram example. <http://www-verimag.imag.fr/~stauch/ArgosCompiler/contracts.html>. 4.0.7
- [11] The Lustre tutorial. <http://www-verimag.imag.fr/~raymond/edu/tp.ps.gz>. 4
- [12] 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. 1.2
- [13] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001. 2
- [14] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992. 1.2

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

$$\begin{aligned} A' \triangleleft asp &= \text{OBS}_A(\text{ND}_A(A) \triangleleft asp), & A \triangleleft asp &= \text{Det}_A(A' \triangleleft asp), \\ G' \triangleleft asp &= \text{OBS}_G(\text{ND}_G(G) \triangleleft asp), \text{ and} & G \triangleleft asp &= \text{Det}_G(G' \triangleleft asp). \end{aligned}$$

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

$$\begin{aligned} P &= (\mathcal{Q}_P, q_{P0}, \mathcal{I}, \mathcal{O}, \mathcal{T}_P), \\ asp &= (\text{PC}, \langle O_{adv}, \text{toInit}, \sigma \rangle), \\ \text{PC} &= (\mathcal{Q}_{\text{PC}}, q_{\text{PC0}}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{\text{PC}}), \\ A &= (\mathcal{Q}_A \cup \{\text{Error}\}, q_{A0}, \mathcal{I} \cup \mathcal{O}, \{\text{err}\}, \mathcal{T}_A), \\ G &= (\mathcal{Q}_G \cup \{\text{Error}\}, q_{G0}, \mathcal{I} \cup \mathcal{O}, \{\text{err}\}, \mathcal{T}_G), \\ P \triangleleft asp &= (\mathcal{Q}_P \times \mathcal{Q}_{\text{PC}}, (q_{P0}, q_{\text{PC0}}), \mathcal{I}, \mathcal{O}, \mathcal{T}_{P \triangleleft asp}), \\ A' \triangleleft asp &= ((\mathcal{Q}_A \times \mathcal{Q}_{\text{PC}}) \cup \{\text{Error}\}, (q_{A0}, q_{\text{PC0}}), \mathcal{I} \cup \mathcal{O}, \{\text{err}\}, \mathcal{T}_{A \triangleleft asp}), \text{ and} \\ G' \triangleleft asp &= ((\mathcal{Q}_G \times \mathcal{Q}_{\text{PC}}) \cup \{\text{Error}\}, (q_{G0}, q_{\text{PC0}}), \mathcal{I} \cup \mathcal{O}, \{\text{err}\}, \mathcal{T}_{G \triangleleft asp}). \end{aligned}$$

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

Induction hypothesis.

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

(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. The existential quantifier before (a_n, pc_n) and (g_n, pc_n) is needed because $A' \triangleleft asp$ and $G' \triangleleft asp$ may be non-deterministic.

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 $O_step_{A \triangleleft asp}(it.ot, n) = \{\text{err}\}$, we are done. Otherwise, $O_step_{A' \triangleleft asp}(it.ot, n) = \emptyset$ holds because of Definition 13, and 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 $\mathcal{T}_{P \triangleleft asp}$, a transition $((a_{n-1}, pc_{n-1}), it(n) \wedge ot(n), \emptyset, (a_n, pc_n))$ in $\mathcal{T}_{A \triangleleft asp}$, and a transition $((g_{n-1}, pc_{n-1}), it(n) \wedge ot(n), \emptyset, (g_n, pc_n))$ in $\mathcal{T}_{G \triangleleft asp}$ with $(g_n, pc_n) \neq \text{Error}$.

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

We have $S_step_{P \triangleleft asp}(it, n) = S_step_P(\sigma, l_\sigma) = p_\sigma$. Let ς be a trace of length l_σ such that $\forall i \leq l_\sigma . \varsigma(i) = O_step_P(\sigma, i)$. 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_σ, pc_σ) (resp. (a_σ, pc_σ)) such that $S_step_G(\sigma, \varsigma, l_\sigma) = g_\sigma$ (resp. $S_step_A(\sigma, \varsigma, l_\sigma) = a_\sigma$) and $S_step_{PC}(\sigma, \varsigma, l_\sigma) = pc_\sigma$. Because p_σ, a_σ and g_σ can be reached with the same trace (σ, ς) (resp. $(\sigma, \varsigma, \epsilon)$ 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, $ot(n) = O_{adv}$, and we have $t_{a' \triangleleft} = ((a_{n-1}, pc_{n-1}), it(n) \wedge O_{adv}, \emptyset, (a_\sigma, pc_\sigma))$, and $t_{g' \triangleleft} = ((g_{n-1}, pc_{n-1}), it(n) \wedge O_{adv}, \emptyset, (g_\sigma, pc_\sigma))$, and thus $(a_\sigma, pc_\sigma) = S_step_{A' \triangleleft asp}(it.ot, n)$ and $(g_\sigma, pc_\sigma) = S_step_{G' \triangleleft asp}(it.ot, n)$. Furthermore, we have $(g_\sigma, pc_\sigma) \neq \text{Error}$, as otherwise $a_\sigma = \text{Error}$ (impossible because of $O_step_{A' \triangleleft asp}(it.ot, n) = \emptyset$), or $(it, ot) \notin \text{Traces}(P)$, by the definition of $P \models (A, G)$.

It follows from the induction hypothesis that

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

and we have $(it.ot, \epsilon) \in \text{Traces}(G' \triangleleft asp) \Rightarrow (it.ot, \epsilon) \in \text{Traces}(G \triangleleft asp)$ by Definition 14. Thus, the theorem follows from the induction hypothesis. \square