Modifying Contracts with Larissa Aspects

David Stauch

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

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.

Keywords: Aspect-oriented programming, Design-by-contract, synchronous languages

1 Introduction

1.1 Synchronous Languages and Aspect-Oriented Programming

Aspect-oriented programming (AOP) offers facilities 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 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

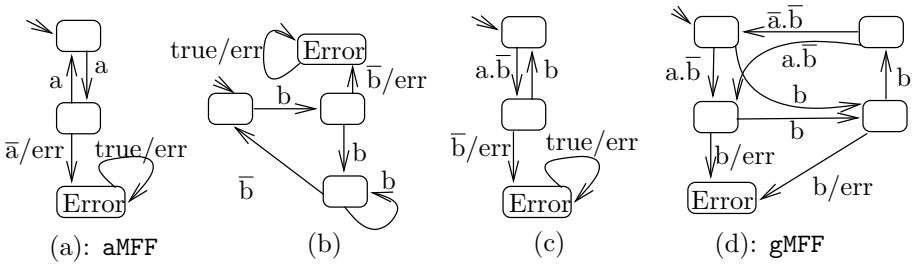


Fig. 1. The contract for the MFF. Notations: in each automaton, the initial state is denoted with a little arrow; the label on transitions are expressed by "`triggering condition / outputs`", e.g. a transition labelled by "`a/b`" is triggered when `a` is true and emits `b`. Negation is expressed with an overbar, and conjunction with a dot. The observers accept all traces that do not lead to state Error.

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.

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. The product of Figure 1(b) and Figure 1(c) is shown in Figure 1(d).

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  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 }
```

Fig. 2. Example of a contract in presence of an AspectJ aspect.

`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 $\backslash result < 11$, 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. An extended version of this paper can be found at [15].

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 2.1 [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: $it : N \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. 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.

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. 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 2.2 [Automaton] An *automaton* $\mathcal{A}$ is a tuple $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where $\mathcal{Q}$ is the set of states, $s_{\text{init}} \in \mathcal{Q}$ 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:

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

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

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

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 $\mathit{Traces}(\mathcal{A})$ is deterministic (resp. complete) (see Definition 2.1). Two automata $\mathcal{A}_1$, $\mathcal{A}_2$ are *trace-equivalent*, noted $\mathcal{A}_1 \sim \mathcal{A}_2$, iff $\mathit{Traces}(\mathcal{A}_1) = \mathit{Traces}(\mathcal{A}_2)$.

Definition 2.3 [Synchronous Product] Let $\mathcal{A}_1 = (\mathcal{Q}_1, s_{\text{init}1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ and $\mathcal{A}_2 = (\mathcal{Q}_2, s_{\text{init}2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ be automata. The *synchronous product* of $\mathcal{A}_1$ and $\mathcal{A}_2$ is the automaton $\mathcal{A}_1 \| \mathcal{A}_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{\text{init}1}, s_{\text{init}2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ 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 2.4 [Encapsulation] Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ 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 $\mathcal{A} \setminus \Gamma = (\mathcal{Q}, s_{\text{init}}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ 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. $\ell^- = \{x \in \mathcal{I} \mid (\overline{x} \wedge \ell) = \ell\}$).

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 programms fulfilling a certain safety property. It is formally defined as follows.

Definition 2.5 [Observer] An observer is an automaton $(\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ 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 $P \models \mathit{obs}$) 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 2.6 is an auxiliary definition, used to formally define contracts in Definition 2.7. ϵ denotes the empty trace.

Definition 2.6 [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 2.7 [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

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

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 2.7 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.

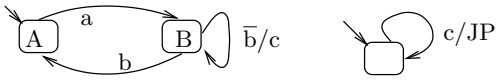
The goal of aspects being precisely to specify such cross-cutting modifications of a program, we proposed an aspect-oriented extension for Argos [1], which allows the modularization of a number of recurrent problems in reactive programs, like the reinitialization. This leads to the definition of a new 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 preserver 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 *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.



(a) base program (b) pointcut

Fig. 3. Example pointcut.

Technically, we perform a parallel 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*.

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

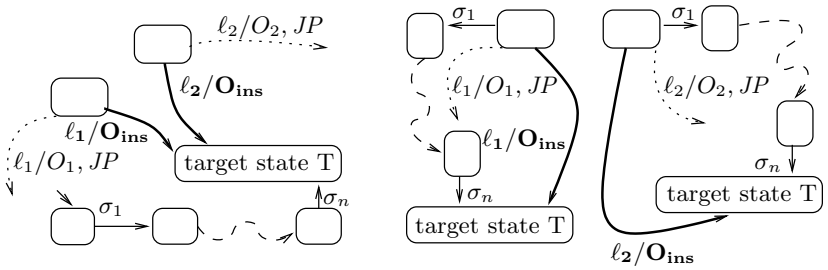
In aspect oriented languages, the advice expresses the modification applied to the base program. In Larissa, we define two types of advice: in the first type, an advice replaces the join point transitions with *advice transitions* pointing to an existing target states; in the second type, an advice introduces 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 3.2 in the following section gives a formal definition for toInit advice. The remainder of this section describes the different kinds of advice informally.

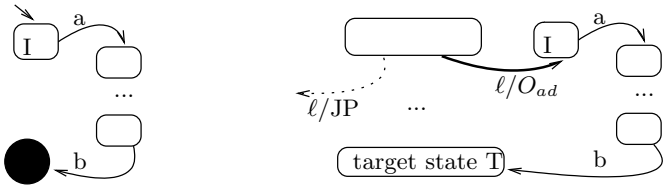
Advice Transitions

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



(a) toInit aspect (b) toCurrent aspect

Fig. 4. Schematic toInit and toCurrent aspects. Advice transitions are in bold, join point transitions are dotted.



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

Fig. 5. Inserting an advice 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 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 *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 $< O_{ad}$,*type*, $\sigma >$, or, with an advice program, a tuple $< O_{ad}$,*type*,σ, $P_{adv} >$, with $type \in \{toCurrent, toInit\}$. An aspect is woven into a program by first determining the join point transitions

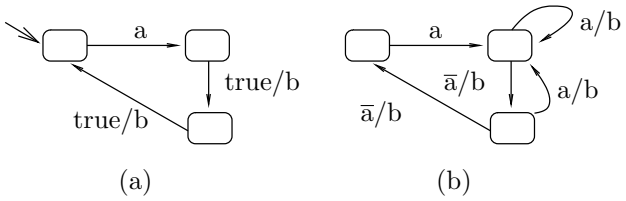


Fig. 6. A possible implementation of the MFF (a), with the retriggerable aspect applied to it (b).

and then weaving the advice.

Definition 2.8 [Aspect weaving] Let P be a program and $asp = (\mathrm{PC}, adv)$ an aspect for P. The weaving of asp on P is defined by

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

2.4.4 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, $< b, toInit, (a) >$) 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. Figure 6(a) shows a sample implementation of the MFF, and Figure 6(b) shows the result of applying `ret` to it.

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 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`,$A \Rightarrow G$). 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

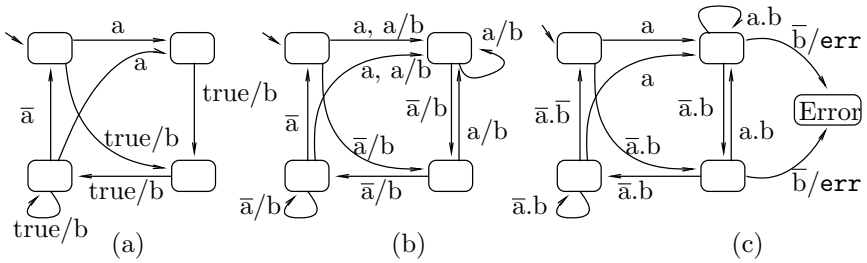


Fig. 7. a: ND_G(gMFF), b: ND_G(gMFF)$\triangleleft$ ret, c: $OBS_G(ND_G$(gMFF)$\triangleleft$ ret).

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.

3.1 Formal Definitions

This paragraph describes the weaving of aspects into contracts in detail, and illustrates it on our running example. First, Definition 3.1 defines the transformation of an observer into a NDA through two functions, one for guarantee observers and one for assumption observers.

Definition 3.1 [Observer to NDA transformation] Let $obs = (\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ 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 $(s, \ell_\mathcal{I} \wedge \ell_\mathcal{O}, \emptyset, s') \in T \Rightarrow (s, \ell_\mathcal{I}, \ell^+_\mathcal{O}, s') \in T_{ND_G}$. $ND_A(obs) = (\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_A})$ 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. As an example, consider the guarantee of the MFF (Figure 1(d)). Its transformation into a NDA is shown in Figure 7(a).

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 (Figure 1(a)) is shown in Figure 8(a).

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 3.2 [toInit weaving for NDA] Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $adv = (O_{adv}, \text{toInit}, \sigma)$ a piece of toInit advice, with $\sigma : [0, ..., \ell_\sigma] \longrightarrow$

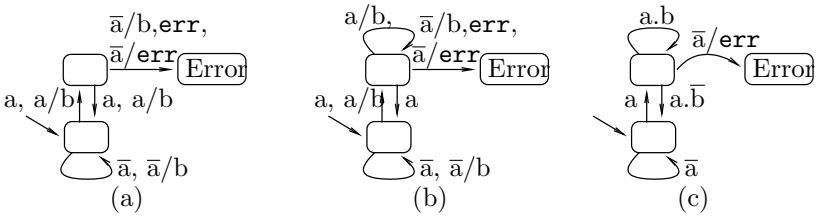


Fig. 8. a: ND_A(aMFF), b: ND_A(aMFF)$\triangleleft$ ret, c: $OBS_A(ND_A(\text{aMFF})\triangleleft \text{ret})$.

$[\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ a finite input trace of length $\ell_\sigma + 1$. Let $TARG = \{s | s = S\_step_{\mathcal{A}}(s_{\text{init}}, \sigma, \ell_\sigma)\}$ be the set of all states reachable with σ. The advice weaving operator $\triangleleft$, weaves adv into $\mathcal{A}$ and returns the automaton $\mathcal{A} \triangleleft adv = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O} \cup O_{adv}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}' \tag{1}$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP \in O\big) \implies \forall targ \in TARG\,.\,(s, \ell, O_{adv}, targ) \in \mathcal{T}' \tag{2}$$

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 $\mathcal{A}$. Figure 7(b) and Figure 8(b) show the NDAs from our example with the retriggerable aspect from Section 2.4.4 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 3.3 [NDA to guarantee transformation] Let $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T)$ be a NDA. $OBS_G(nd) = (\mathcal{Q} \cup \{\text{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T' \cup T'')$ defines an observer, where T' and T'' are defined by

$$(s, \ell, o, s') \in T \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O} \setminus o}}, \emptyset, s') \in T' \tag{3}$$
$$\begin{aligned}&(s, \ell, \emptyset, s') \notin T' \wedge s \in \mathcal{Q} \wedge \ell \text{ is a complete monomial over } \mathcal{I} \cup \mathcal{O} \\ &\qquad \Rightarrow (s, \ell, \{\texttt{err}\}, \text{Error}) \in T''\end{aligned} \tag{4}$$

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

Definition 3.4 [NDA to assumption transformation] Let $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O} \cup \{\texttt{err}\}, T)$ be a NDA. $OBS_A(nd) = (\mathcal{Q}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T')$ defines an observer, where T' is defined by

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

Figure 7(c) and Figure 8(c) show the NDAs from our example transformed back into observers. As expected, the obtained guarantee in Figure 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.

Definition 3.5 [Assumption Determinization] Let M be a NDA with outputs $\{\texttt{err}\}$. $Det_A(M)$ is a deterministic automaton such that

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

Definition 3.6 [Guarantee Determinization] Let M be a NDA with outputs $\{\texttt{err}\}$. $Det_G(M)$ is a deterministic automaton such that

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

We can now state the following theorem. See [15] for a proof.

Theorem 3.7 *Let P be a program and let (A, G) be a contract. Then,*

$$P \models (A, G) \\ \Rightarrow P \triangleleft \mathit{asp} \models (\mathit{Det}_A(\mathit{OBS}_A(\mathit{ND}_A(A) \triangleleft \mathit{asp})), \mathit{Det}_G(\mathit{OBS}_G(\mathit{ND}_G(G) \triangleleft \mathit{asp})))$$

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.

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. Figure 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 Figure 9. They are calculated from the original inputs, by a program given in [15].

It is important for the safety of the passengers that the doors are never open outside a station. We give a contract for the door controller, which focuses on this

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

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

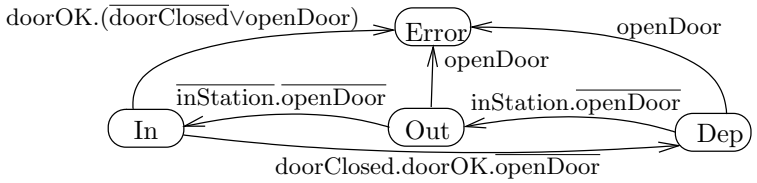


Fig. 10. The guarantee of the contract of the controller.

property. The guarantee of the contract is shown in Figure 10, 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, which requires that the door behaves correctly (e.g., the door only opens if openDoor has been emitted). It is given in [15], along with an implementation of the controller.

To formally verify that a tram door is always closed outside a station, we develop a model that describes the possible behavior of the physical environment of the controller, i.e. the door and the tramway. These models are expressed as Argos observers, and are given in [15]. We then prove that the controller satisfies the contract, and that the contract in the environment never violates the safety property.

4.1 Adding The Gangway

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 $\mathrm{PC}_{\mathrm{ext}}$ of the extension aspect selects all transitions where $openDoor.doorReq.doorClosed.\overline{gwOut}$ is true, and the pointcut $\mathrm{PC}_{\mathrm{ret}}$ of the retraction aspect 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 Figure 11. The extension aspect is specified by $(\mathrm{PC}_{\mathrm{ext}}, < \{\}, toCurrent, (), I_{\mathrm{ext}} >)$, and the retraction aspect by $(\mathrm{PC}_{\mathrm{ret}}, < \{retractGW\}, toCurrent, (), I_{\mathrm{ret}} >)$.

We want to check that the new controller still verifies the safety property from

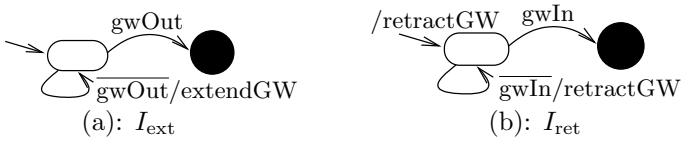


Fig. 11. Inserted automata for the extension (a) and the retraction (b) aspect.

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. Therefore, we weave the aspects into the contract, and thus obtain a new contract that holds for controller with the aspects. Finally, we check then that the environment, to which we added a model of the gangway, satisfies the new assumption, and that the new guarantee satisfies the safety requirements in the environment.

An alternative to this modular approach is to verify directly that the sample controller with the aspects does not violate the given safety properties. 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 [9]. The source code of the door controller example is available at [10]. 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 Related Work

Goldman and Katz [5] 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 [4] 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 [3] 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.

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. This approach has several advantages. First, aspects can be checked against contracts even if the

[1] Experiments were conducted on an Intel Pentium 4 with 2.4GHz and 1 Gigabyte RAM.

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 for larger program to be verified, as indicates the example in Section 4.

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 $t \in \mathit{Traces}(C \triangleleft \mathit{asp})$, there exists a program P s.t. $P \models C$ and $t \in \mathit{Traces}(P \triangleleft \mathit{asp})$. 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 $C' \triangleleft \mathit{asp} \models C$? 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.

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

[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.

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

[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.

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

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

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

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

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

[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.

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

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

[15] D. Stauch. Modifying contracts with Larissa aspects. Technical Report TR-2006-10, Verimag, Dec. 2006.