

Aspect-Oriented Programming for Reactive Systems: Larissa, a Proposal in the Synchronous Framework

K. Altisen and F. Maraninchi and D. Stauch

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

Abstract

Aspect-Oriented Programming (AOP) has emerged recently as a language concept for expressing cross-cutting concerns, mainly in object-oriented software. Since then, the concept has been applied to a wide variety of other contexts. In this paper, we explore some cross-cutting concerns for parallel programs of reactive systems: we propose an aspect language, Larissa, and a weaving mechanism, in a core language based on parallel communicating Mealy machines.

1 Introduction

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

From our point of view, the AOP community is divided into two families: (1) the first family is guided by the needs on tracing, profiling, debugging: aspects are used to describe functionalities that do not modify the behavior of the original program. Such an aspect should be allowed to access/observe every entity (variables, functions, ...) in the program, provided that it does not interfere with the execution of the program. The way it refers to the entities of the program may be syntactic, or even lexical, but its role is not intrusive. (2) In the second family, aspects aim at *modifying* the behavior of a program. The way aspects are allowed to interfere with the program behavior,

3 February 2006

and with each other, has to be clearly defined and limited. The definition of an aspect needs to be more *semantic*. Indeed, one would intuitively expect that whenever two programs P and P' are semantically equivalent (i.e. behave the same), applying such an aspect to P gives the same result (semantically speaking) as applying it to P' , even if they have very different syntactic forms. Of course, which entities of the program the aspect is allowed to refer to, and how far the weaving mechanism can change the execution of the program, are deeply linked to the definition of the semantic equivalence.

In this paper, we are interested in the notion of cross-cutting concerns for reactive systems. The existing structure of the program we want to cross-cut is based on parallel composition. Since reactive systems very often execute in critical contexts, it is very important to understand their behavior, so we use formally defined languages. That is why we study aspects with the point of view number (2): we need some definition of an aspect and some weaving mechanism that guarantee the preservation of a semantic equivalence.

An important question is the choice of the base *language* on which we will define aspects. We chose a very simple set of operators, capturing the main ideas behind *synchronous languages*. The reasons are the following: first, synchronous languages and formalisms share a very clean and simple definition of parallelism, and it is a good starting point for studying aspects that crosscut a parallel structure. Second, the need for *aspects* clearly appeared in synchronous programming.

1.1 The Discrete View on Reactive Systems

A reactive system is a computer system embedded into an *environment* made of physical phenomena, human beings and other computers. It has *input signals* coming from the environment (sensors for the physical phenomena, normal inputs from the human beings, incoming communications from other computers) and *output signals* sent to this environment (actuators, normal outputs, outgoing communications). *Reactive* means that we have to consider the system together with its environment, in order to specify and program its behaviour. Indeed, an output issued at some point in time may have some influence on the inputs received later, via the reaction of the environment.

Of course, the environment (at least the physical part of it) is intrinsically *continuous*. But the reactive system itself is a computer system, hence it has a discrete behaviour. Programming reactive systems imposes a *discrete* view of time: time can be viewed as divided into *instants*. All inputs that occur in the same instant are considered to be simultaneous. The connection between the reactive system and its environment faces sampling problems, but the

implementation of the *reactive kernel* may be designed with this discrete view.

1.2 Programming Reactive Systems: the Synchronous Approach

Over the past ten years, the family of synchronous languages [4,13] has been very successful in offering domain-specific, formally defined languages and programming environments for safety-critical reactive systems. The family is composed of *data-flow* languages (Lustre, Signal) and imperative languages (Esterel, Argos).

All these languages, even if they may seem to have very different styles, share a common semantical basis. As stated above, time is discrete and divided into instants. In each of these instants, a program reacts to *inputs* by sending *outputs* and updating its internal memory. Such a reaction is *atomic*: the system does not read inputs while computing outputs and updating its memory. All the synchronous languages may be compiled into very simple programs of the following form:

```
initialize memory m ;
while (true) {
  read inputs i ;
  compute outputs o (depending on i and m) ;
  update memory m (depending on m and i)
  emit outputs o ; }
```

The motivation for designing high level languages that can be compiled into this very simple scheme is the need for *parallelism*. A complex reactive system is very hard to describe directly as a simple loop. As soon as it has to take several inputs into account, it is likely to be composed of some “parallel” activities. For instance, it could have to send one output “A” each time it has seen two inputs “a” and, during the same time to send one “B” each time it has seen three “b”. In all synchronous languages, this behavior can be described as the parallel composition of the two activities. The compiler then produces a piece of sequential code, relying on some ordering of the tests on inputs, but the user does not have to think in terms of this low level ordering. Parallelism can be considered as the main structure of reactive programs, regardless of the language used. Any notion of aspect will naturally cross-cut *this* structure.

When the parallel activities are not independent, the synchronous languages provide a very powerful synchronization mechanism called *synchronous broadcast*. For instance, consider the two parallel activities : “send an 'A' every two 'a' ” and “send a 'B' every three 'A' ”; the activities synchronize on 'A' which is emitted and received during the same instant. The main idea behind this construct, often referred to as the “synchrony hypothesis”, is the following: it

allows the use of parallel components instead of sequential code when it is convenient from the logical point of view, *without* influence on the code produced. This is the case because the synchronization between parallel components is entirely *compiled* into sequential code where it is no longer visible.

One of the advantages of the synchronous broadcast is its *asymmetry*: sending is non blocking, and the sender does not have to know the number of potential listeners. This allows to use so-called *observers* [15], i.e., *programs* that specify safety properties (in the sense of Lamport [22]) and that are put in parallel with the system observed, without changing its behavior. These observers may be used as a kind of temporal logic to specify properties to be checked, or as oracles in a test framework. In the context of aspects, observers seem a very good candidate for defining dynamic join points.

1.3 Aspects for Reactive Systems

In spite of the powerful parallel composition and synchronization mechanism shared by all synchronous languages, some recurring problems appear when programming complex reactive systems.

For instance, in some languages, something as simple as modifying an existing program in such a way that it becomes reinitializable when some new event occurs, cannot be done without introducing some new code everywhere in the parallel components. In Esterel, the need for reinitializable subprograms has led to the definition of a dedicated construct, but adding a dedicated construct as soon as a similar need appears is a never-ending process.

In this paper, we investigate a notion of aspects for reactive systems programmed with synchronous languages. Reinitialization will be a very simple case of it, which can be specified informally as follows: in any state, whenever a special signal “reset” occurs, do not behave as specified by the program, but rather go to its initial state.

In more general cases, we need to specify *where in the program* the aspect should have some effect. As we do not want to talk about the program syntactically, we look at its execution traces and we specify *when* the aspect should have some effect. This leads to specifying dynamic join points which are defined w.r.t. the *history* of the system’s behavior. This is intrinsic to reactive systems, whose behavior highly depends on the history of inputs. For instance, we could need to change the behavior of the system when some “Alarm” occurs, but only if it is after an occurrence of another event.

Let us compare these needs to a `cflow`-like mechanism. In AspectJ [19], `cflow` is used as an abstraction of the stack, and may specify dynamic join points.

For instance, one may say that he would like to do something in function f , but only if it has been called from function g . The information needed for checking this condition is indeed computed by any runtime environment, and present in the stack at any time. But a specification like “do something in function f , but only when called *after* function g ” is not implementable with the history abstraction contained in the stack. Other related works on dynamic join points are exposed in section 6.

In our framework, it is natural to use observers to specify dynamic join points. It means there is no limitation on the history information an aspect may need as far as it is bounded.

1.4 Approach and Contributions

We consider that AOP techniques and tools have demonstrated their interest in various domains. We suspect that some of the recurring problems people have encountered when programming complex reactive systems (and making the programs evolve) can be better understood and dealt with if we clearly consider them as aspects. Since we are interested mainly in *critical* systems, we care about formal semantics, and therefore would like our notion of aspect to be more “semantical” than “syntactical”. We will concentrate on the preservation of a behavioral equivalence.

To investigate these ideas further, we propose to define and implement a particular notion of aspects for a formally defined language, in the context of reactive systems. This means:

Choosing a base synchronous language for reactive systems. We choose to work on a formalism whose structure is intermediate between the structure of high level languages (because it would be too specific) and the simple flat automata representing the reactive behaviors (because it would be degenerated). Our core language is made of Mealy machines composed in parallel, and communicating via the synchronous broadcast. This is enough to study cross-cutting of a parallel structure, and to use observers freely. The simplest way of expressing the semantics of such a language is to consider that any composition of automata can be “flattened” into a single automaton.

Proposing a language of aspects and a weaving mechanism, in a somewhat “minimal” way: first, we should avoid introducing aspect-oriented features for the behavior transformations that were already feasible without them; second, we would like to identify the “basic blocks” needed to define aspects for reactive systems. The weaving mechanism is static (it can be part of a compiler), but the specification of join points may refer to the dynamics of the system’s behavior. When using observers, the language of join points is

the programming language itself.

Proving that aspect weaving preserves the usual behavioral equivalence used in the context of reactive systems.

Implementing this language and this weaving mechanism in an existing compiler.

As a by-product, this work will also give some hints on the desirable properties of an aspect-oriented mechanism, when introduced in a formally defined language. But we do not aim at formalizing this “meta” notion, nor at proposing a list of these desirable properties.

1.5 Structure of the paper

Some of the ideas in this paper were already sketched in [1], but without formalization. The structure of the sequel is as follows: section 2 describes the simple language on which we will define aspects; section 3 gives an introductory example: it explains informally the notion of aspect we deal with and the weaving mechanism; section 4 gives formal definitions for: the language semantics (section 4.1), the aspect language Larissa and the weaving mechanism (section 4.2); section 5 presents the implementation and extensions; section 6 explores related works; section 7 concludes.

2 A Simple Language for Reactive Systems

In this section, we describe a very simple language for reactive systems, meant to illustrate the main principles of synchronous programming and languages. It is a restriction of Argos [25]. We also give an example program.

2.1 Mealy Automata

The language we describe allows the use of explicit automata to describe reactive kernels. Since the size of explicit automata grows rapidly with the complexity of systems, we also provide a parallel composition construct.

Fig. 1 gives a Mealy automaton with Boolean guards and actions, which implements the following behaviour: the system has three inputs x , y and z that may occur simultaneously, and one output a . a is emitted whenever the system has observed an occurrence of x (not necessarily alone) followed by

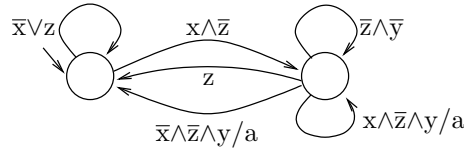


Fig. 1. A Mealy machine.

an occurrence of y (not necessarily alone either), with no occurrence of z in between.

Transitions are labeled by a Boolean condition on input signals, and a set of emitted signals. We use the concrete syntax: `condition / emitted signals`. The automaton of fig. 1 is both *deterministic* and *complete*, i.e., for any state q , for any possible configuration C of the inputs, there is exactly one transition sourced in q whose condition is true for C . An automaton like the one in fig. 1 is very easy to compile into sequential code, of the form shown in the introduction.

2.2 Parallel composition and Encapsulation

A reactive system can sometimes be difficult to design directly as a flat automaton, because of the number of its states and transitions. The parallel composition may then be used.

In the language we present here, we clearly distinguish between: the *parallel* composition of machines, that reflect exactly the intersection of their behaviors; and the *encapsulation*, that forces two machines to synchronize on some signal. In practice, the two operators are used in conjunction as illustrated by fig. 2. Note that we suppose all automata in the sequel to be implicitly complete: if a state has no transition for some input valuations, we suppose that there is a self-loop transition with these valuations as triggering condition and no outputs. The system described in fig. 2 is a modulo 8 counter, counting the occurrences of the signal a , and emitting hi every 8 a 's. It could be described directly as a 8-states Mealy machine (fig. 2(a)), or as the result of a parallel composition between 3 small automata that represent the bits: *bit0* counts a modulo 2 and emits a carry $c0$ towards *bit1*. *bit1* counts $c0$ modulo 2 and emits a carry $c1$ towards *bit2*. *bit2* counts $c1$ modulo 2 and emits the output hi . The scope of the signal $c0$ is limited to the parallel composition of *bit0* and *bit1*. The scope of $c1$ is limited to the parallel composition of *bit2* with the composition of *bit0* and *bit1*.

The parallel composition alone is just the synchronous product of two machines. To illustrate this, we draw part of the product of *bit0* and *bit1* (fig. 2(b)), without the encapsulation by $c0$. From the state BC , for instance,

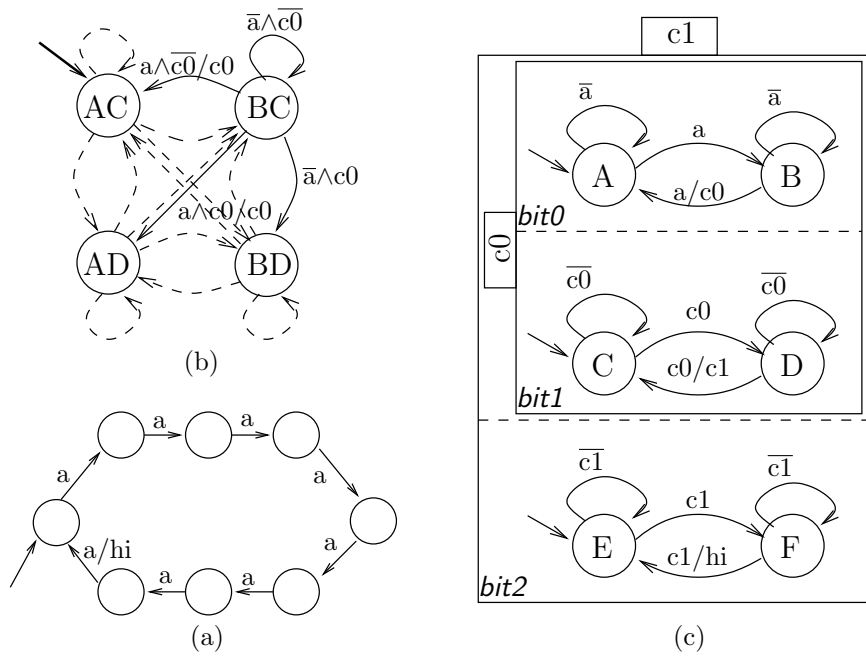


Fig. 2. A modulo-8 counter.

all possible pairs of transitions are present: the loop $\bar{a}\wedge\bar{c}0$, and the transitions $\bar{a}\wedge c0$ to BD, $a\wedge\bar{c}0/c0$ to AC, $a\wedge c0/c0$ to AD.

The encapsulation by $c0$ then forces the synchronization according to the signal $c0$ that is an input of one component, and an output of the other one. The idea is to express the fact that, if $c0$ is emitted by a component, then the other one cannot ignore it; conversely, a component cannot react as if $c0$ was present, if the other one does not emit it. On the example of the two bits, we have to remove the transitions $\bar{a}\wedge c0$ (because it corresponds to the second component reacting to $c0$, while the first one does not emit it) and $a\wedge\bar{c}0/c0$ because it corresponds to the first component emitting $c0$ while the second ignores it. Conversely, we will keep the transition $a\wedge c0/c0$ from BC to AD, because it corresponds to a correct synchronization: *bit0* emits $c0$ and *bit1* reacts to it. In the result, the local signal $c0$ will be hidden.

2.3 Determinism and Completeness

Reactive systems designed with this language are programs to be implemented. They need to be complete and deterministic since generating code for non-deterministic or non-complete systems is meaningless. This means respectively: the system accepts any given sequence of inputs, and two executions with the same sequence of inputs give the same outputs. To obtain those properties, dealing with individual deterministic and complete Mealy machines is often sufficient, but not always. Indeed, the parallel composition preserves

both, but there are some particular cases when the encapsulation operator does not preserve determinism nor completeness of its operand. Those cases, studied under the name of “causality problem” [5], can appear only if two parallel individual Mealy machines communicate in both directions, in the same instant; for instance, if the first emits a 'b' when it receives an 'a' and the second emits an 'a' when it receives a 'b', this means that 'a' and 'b' have the same value but it could non-deterministically be true or false. In all synchronous languages, a detection mechanism is defined for these cases. It ranges from pure syntactic restrictions to more sophisticated semantic analysis, including state reachability.

In the rest of the paper, we will define aspects as program transformations and show that aspect weaving may be considered as a new operator. We will show that this new operator preserves both determinism and completeness.

2.4 Modular Programming

A complete language based on explicit automata would also provide a construct dedicated to build *hierarchies* of states, as in Statecharts [16]. However, the parallel composition and the encapsulation are already very expressive, and we chose not to include such a hierarchic operator in our base language.

As an example for modular programming, imagine a system with three inputs x , y and z that may occur simultaneously, and one output hi . hi is emitted every eighth occurrence of the following situation: whenever the system has observed an occurrence of x (not necessarily alone) followed by an occurrence of y (not necessarily alone either), with no occurrence of z in between. This new system is simply the parallel composition of the modulo-8 counter with the detector of fig. 1, encapsulated according to the signal a .

A program in the language is thus a set of Mealy machines composed with the parallel composition and the encapsulation operators. Its semantics is obtained when flattening the operators as shown in the modulo-8 counter example (fig. 2).

3 Introductory example

This section illustrates the whole approach on two examples. They are extracted from the same case study, namely a juice processing plant [10]. The plant is divided into three departments. The first one aims at producing the juice, the second one pasteurizes it and the last one packages it. Our examples

focus on the high level controls of the production process (section 3.2) and of the pasteurization (section 3.1).

For each of them, we first describe the physical system to be controlled, and then its controller with focus on its inputs and outputs. Inputs of the controller are mainly information coming from the plant (e.g. signals from sensors) and outputs of the controller are commands that are sent to the plant (e.g. close a valve). The controller is the program we work on: its interface, namely its inputs and its outputs, is precisely known, but the way it is programmed is not known. We just know that its role is to control the plant with respect to the inputs it receives and by sending commands. The second part of the example describes a functionality to be added to a controller. We there introduce our notion of aspect and show how to use it to express the new functionality. The third part of each example shows a sample controller and how the above functionality is added by weaving the aspect. Giving a *sample* controller *after* having specified entirely the aspects aims at showing that our approach is completely *oblivious* [11]: nothing has been foreseen in the program to weave the aspects. Moreover, our approach is also black-box: the aspect has not been tailored for a given implementation of the program.

3.1 *The pasteurizer controller*

By means of an example aspect for the pasteurizer controller, this section introduces our notion of aspect.

3.1.1 *The physical machine*

A simplified view of the pasteurization department is shown in fig. 3. The juice to be pasteurized comes from the input buffer tank. When the valve is open, the juice continuously flows to the pasteurizer machine where it is heated and then quickly cooled down. When leaving the pasteurizer, it flows to the output buffer tank. After one cycle of pasteurization, the pasteurizer machine and the buffer tanks must be cleaned. This is done by the cleaner machine which is not detailed here.

3.1.2 *The controller specification*

A user interface panel contains two buttons, one to ask for cleaning, the other for pasteurizing. When the machine is idle, pressing the `clean` button begins a cleaning cycle and pressing the `past` button starts the pasteurizer. The pasteurizer cannot be used while the cleaner machine is working. The machine has various sensors and actuators. The sensors transmit Boolean signals to

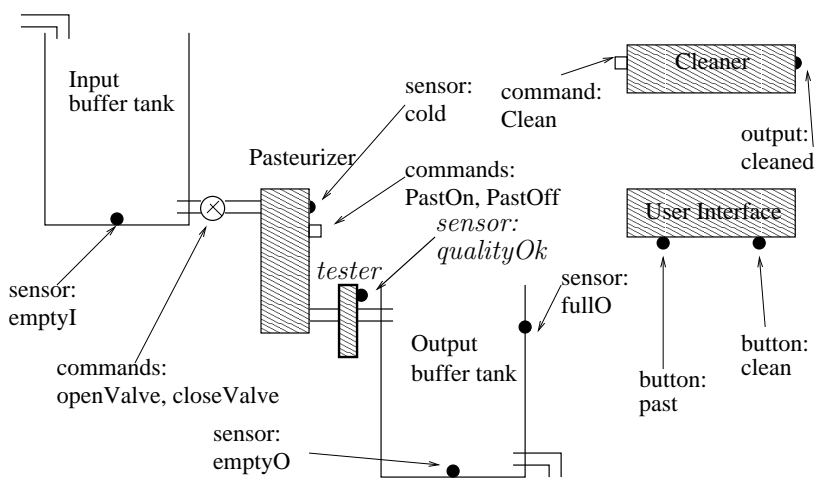


Fig. 3. A simplified view of the pasteurization department

	Buttons and sensors:		Commands:
clean	clean the pasteurizer (button)	Clean	make the cleaner work
past	pasteurize juice (button)	PastOn	switch the pasteurizer on
emptyI	the input buffer tank is empty	PastOff	switch the pasteurizer off
emptyO	the output buffer tank is empty	openValve	open the valve connecting the input buffer tank and the pasteurizer
fullO	the output buffer tank is full	closeValve	close the valve
cold	the pasteurizer is cold		
cleaned	the cleaner has finished cleaning		

Fig. 4. Meaning of the sensors, buttons and commands.

the controller; the actuators receive signals sent by the controller. Thus the *inputs* of the controller are the signals from the sensors and buttons and its *outputs* are the commands to the actuators. The signals and the commands are summarized in fig. 4.

We assume we have a program for the controller, written in the language presented in section 2. This means that: 1) inputs and outputs of the program are the inputs and outputs of the controller; 2) the program is designed as a set of Mealy machines composed with the parallel product and the encapsulation; 3) the transitions of the whole program (compiled into a flat automaton) are of the form C/A where C is a Boolean expression on inputs of the controller and A is a subset of the outputs of the controller.

3.1.3 Modifying the controller to add a quality tester

Suppose we now want to add a quality tester, *Past*, to the department. It will be placed between the pasteurizer and the output buffer tank to test the quality of the juice (in *italics* in fig. 3). It is a sophisticated sensor whose signal *qualityOk* is false whenever a problem is detected and true otherwise.

When a problem of quality is detected, we should stop the pasteurization process, and then clean the machines, before being able to restart another pasteurization process. This new behavior has to be added to the normal behavior of the controller. We can specify it precisely as follows: “*whenever pasteurizing, if `qualityOk` is false, then stop the pasteurization process and start cleaning*”.

If we think of the internal structure of the existing controller program (the explicit states and transitions of the flat automaton), the new behavior could be implemented in the following way: 1) identify the set X of the states that correspond to the condition “*whenever pasteurizing*”; 2) add a transition from each state $x \in X$ to the state that corresponds to the cleaner beginning to work; those transitions have to be triggered by “*if `qualityOk` is false*” and should meanwhile “*stop the pasteurization process*”. This kind of program transformation is quite tricky, and it seems that we have to know the existing program in full detail to be able to perform it.

In the sequel, we propose to use the specification “*whenever pasteurizing, if `qualityOk` is false, then stop the pasteurization process and start cleaning*” as the definition of an aspect, and to weave it automatically in the existing controller. We will show that it requires that we know the specification of the controller (section 3.1.2), but not its internal structure.

3.1.4 A notion of aspect

To perform the kind of transformation we mentioned for the controller, we need to specify: (1) the transitions to be added, this corresponds to the *advice* of the aspect; (2) where to add them (source states of the added transitions), this corresponds to the *pointcut* of the aspect.

Specifying the advice. The transitions to be added have to be defined by their triggering condition, their outputs, and their target state. In the example, the controller has to clean the machines when `qualityOk` is false (this is the triggering condition). Meanwhile, it has to emit the outputs `Clean` (to start cleaning), `PastOff` and `closeValve` (to stop the pasteurization process). To define the target state of the added transitions, we use a finite sequence of input values σ . This sequence uniquely determines a state in the program because it is complete and deterministic: it is the state that would be reached by executing the sequence of inputs from the initial state of the program. We call this kind of advice a *toInit advice*: all added transitions go to the same target state identified by the finite input trace σ executed from the initial state. In the example, we need to specify the state where the cleaner begins to work. As specified in section 3.1.2, when the machine is idle (as it is in its initial state), pushing the user button `clean` starts the cleaner. Hence the

state where the cleaner begins to work can be reached from the initial state in one step if the input `clean` is true, i.e. by the finite input trace $\sigma = (\text{clean})$.

Specifying the join points by describing a *pointcut*. In our case, the join points are the source states where the advice transitions are added. Again, we have to specify states in a system whose internal structure is not known. Using the same trick as above (a finite sequence of inputs, to be played from the initial state) cannot be used here, because it would break the preservation of the behavior equivalence (two equivalent programs should still behave the same after the weaving of the same aspect in each of them). Indeed, specifying states by the execution of a finite input trace from the initial state, would be able to distinguish equivalent states, if they are reachable by input sequences of different lengths. For example, the two automata $A_1 = \{q \xrightarrow{a/b} q\}$ and $A_2 = \{q \xrightarrow{a/b} q', q' \xrightarrow{a/b} q\}$ are equivalent; they both represent the traces where $a = b$ in every instant. But the states q and q' of A_2 are distinguished by the finite input trace $\sigma' = (a)$: adding a transition from the state reached by σ' in A_1 and in A_2 leads to non equivalent automata. (Notice that distinguishing equivalent states by a finite trace was not problematic for the selection of *target* states, because the notion of state equivalence is based on the *future* execution from these states, which does not change when adding transitions to these states.)

For Larissa, we choose to specify join points with a program, called the *pointcut program*. Technically, it is a synchronous observer of the original program, composed in parallel with it, that may observe its inputs and outputs, and that emits a single fresh output JP each time a join point is reached.

In the example, the states X where we need to add transitions are the states in which the pasteurizer is working. The pointcut program will then listen to the inputs and outputs of the program in order to deduce when it is in a state of X . Here, the commands issued to the actuators are sufficient, because the controller enters X whenever it emits the command `PastOn` and it leaves X when it emits `PastOff`. The automaton in fig. 5 emits JP whenever it is in X , thus marking the states in X as join points.

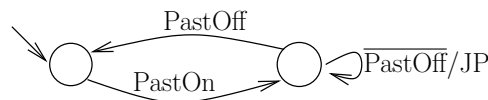


Fig. 5. Pointcut program for the quality tester.

3.1.5 A sample controller.

Fig. 6 shows a sample controller. It can either be cleaning or pasteurizing. When pasteurizing, if the input buffer tank is empty or the output buffer

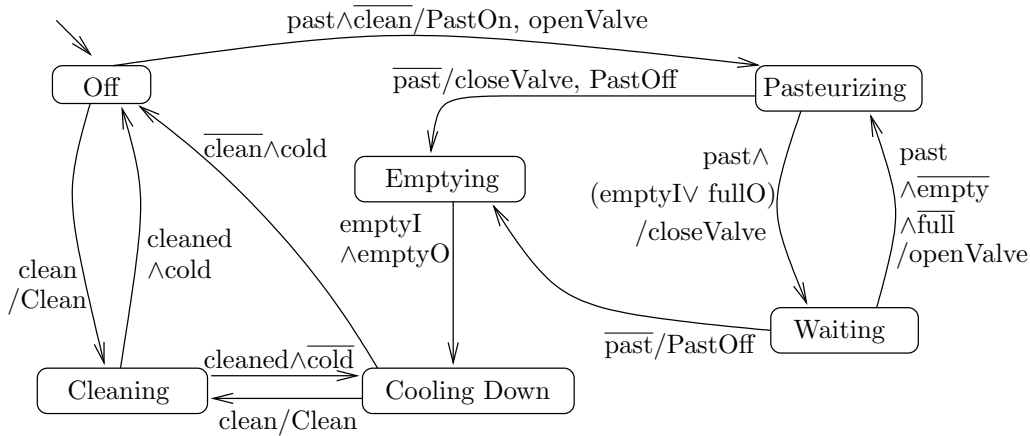


Fig. 6. A sample pasteurizer controller.

tank is full, the process is suspended for a while by commuting the valve. The process is interrupted when the signal `past` becomes false. Before returning to the `Off` state, the controller has to empty the buffer tanks (state `Emptying`) and wait for the pasteurizer to be cold (state `Cooling Down`). The pasteurizer can be cleaned when cooling down.

Notice that the controller is deterministic: when exiting, e.g. the state `Off`, one can go either to `Cleaning` if `clean` is true or to `Pasteurizing` if `past` is true, if `past` and `clean` are both true, priority is given to `clean`.

This controller explicitly contains states that represent the requirement “between an occurrence of `PastOn` and the following occurrence of `PastOff`”, namely the states `Pasteurizing` and `Waiting`. These are the join points identified by the pointcut program fig. 5. The sequence σ selects the state reached from the initial state when `clean` is true, namely `Cleaning`: this is the target state of the advice transitions. Finally the weaving aspect mechanism will add transitions from the `Pasteurizing` and `Waiting` states to the `Cleaning` state. These transitions will be labelled by `qualityOk/PastOff`, `closeValve`, `Clean`.

Weaving an aspect into the controller has modified its interface: the input `qualityOk` has been added. The added transitions are triggered by `qualityOk`. If the other transitions sourced in the same state are unchanged the automaton is non deterministic. To recover determinism, we reinforce their condition by `qualityOk`. The woven controller is partially given in fig. 7.

3.2 Blender controller

In the first example (the pasteurizer), aspect weaving modifies the program by adding transitions to a given point in the program, which is entirely specified

Added transitions:		
Pasteurizing	$\overline{\text{qualityOk}} / \text{PastOff, closeValve, Clean}$	Cleaning
Waiting	$\overline{\text{qualityOk}} / \text{PastOff, closeValve, Clean}$	Cleaning
Modified transitions:		
Pasteurizing	$\text{past} \wedge (\text{emptyI} \vee \text{fullO}) \wedge \text{qualityOk} / \text{closeValve}$	Waiting
Pasteurizing	$\overline{\text{past}} \wedge \text{qualityOk} / \text{closeValve, PastOff}$	Emptying
Waiting	$\text{past} \wedge \overline{\text{emptyI}} \wedge \overline{\text{fullO}} \wedge \text{qualityOk} / \text{openValve}$	Pasteurizing
Waiting	$\overline{\text{past}} \wedge \text{qualityOk} / \text{PastOff}$	Emptying

Fig. 7. The woven controller: modified and added transitions.

by a finite input trace from the initial state (and thus is independent of a particular execution). This section gives an example where an aspect should be able to make a program go *backwards* in a particular execution.

3.2.1 The physical machine and its specification

The *blender* is the unit of the production process where the various ingredients of a juice are mixed. The blender is connected to a manifold, which provides different juice concentrates, namely for apple, orange, and tomato juice. The blender mixes one of these juice concentrates with water in a tank. The different juices may have different water/juice ratios. Once the tank is full, the product is pumped to the next processing unit, the pasteurizer.

The blender provides a user interface with a command for each juice. To start the blender, one must choose a juice. The blender then tells the manifold to connect to the corresponding juice concentrate. When the manifold is connected, the blender starts the production of the juice. Once the tank is full, the blender pumps its content to the next processing unit, and waits for a command to start a new production. The interface of the blender is detailed in fig. 8.

	Buttons and sensors:		Commands:
A	produce apple juice (button)	cncA	connect to apple juice concentrate
T	produce tomato juice (button)	cncT	connect to tomato juice concentrate
O	produce orange juice (button)	cncO	connect to orange juice concentrate
cncd	the manifold has connected to a pipe	addW	add water to the tank
full	the tank is full	addJC	add fruit juice concentrate to the tank
empty	the tank is empty	pump	pump content of the tank to the next unit
tick	a timer signal; true every n seconds		

Fig. 8. The blender interface.

3.2.2 *Modifying the blender controller*

The blender controller has the disadvantage that the production has to be manually restarted each time the tank is full, and that the manifold reconnects each time a process restarts (reconnection takes time), even though this would not be necessary when the choice of juice has not changed. Therefore, we want to add a new command `restart` that tells the blender to restart the current production after the tank has been emptied. When `restart` is true, we do not want the manifold to reconnect, but restart the production of the same juice directly.

3.2.3 *Recovery aspects*

We specify the additional functionality as follows: “when `restart` is true after the tank has been emptied, go back to the point where the production of the current juice started”. As in the previous example, this aspect will add advice transitions from join points.

Specifying the triggering condition. The aspect specification requires the tank to be empty, i.e. the input `empty` must be true when the aspect is activated. We can state this directly by setting the triggering condition to `restart^empty`.

Specifying join points. As before, we use a pointcut program to identify the join points, i.e. the states where the triggering condition should apply and restart the current production. Here, the join points are the states where the tank is emptying, i.e. when `pump` is true. The pointcut program is shown in fig. 9(b). Together with `empty` in the triggering condition, this ensures that the tank has just been completely emptied.

Specifying the “recovery” advice. We have to specify where to go, *backwards*. We cannot simply use a trace to be played backwards, because programs are usually not deterministic in this direction. For example, in the sample pasteurizer controller, if we want to go backwards from the `Emptying` state by `past`, there is no way to know if we should take the transition to `Waiting` or to `Pasteurizing`.

Therefore, we propose a different mechanism to specify where to jump backwards. Some *recovery states* are defined globally and the program will only be able to return to the last recovery state it was in. To specify the set of recovery states, the same mechanism as for join points is used: a *recovery program* observes the inputs and outputs of the program and emits a single fresh output `REC` when a recovery state is reached.

Restarting the juice processing should bring the program to the point where it

started producing juice, i.e. just after `cncted` was true for the first time after we told the manifold to connect to a juice. The recovery program is shown fig. 9(a).

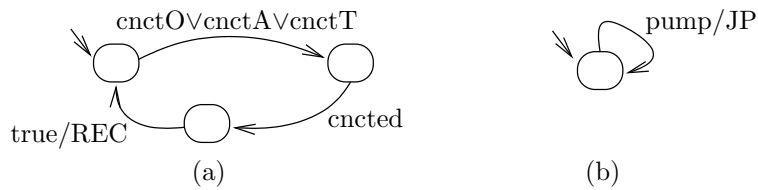


Fig. 9. The recovery (a) and the pointcut program (b) for the recovery aspect.

Note that, again, we defined the aspect in a completely oblivious way w.r.t. the actual implementation of the blender controller; the knowledge of its specification was sufficient.

3.2.4 A sample blender controller

Fig. 10 shows a sample implementation. From the **Start** state, a connection to the manifold is made. Then the juice is “cooked” (water+juice for orange, water+2*juice for tomato and apple) and then pumped to the next processing unit, before the controller goes back to the **Start** state.

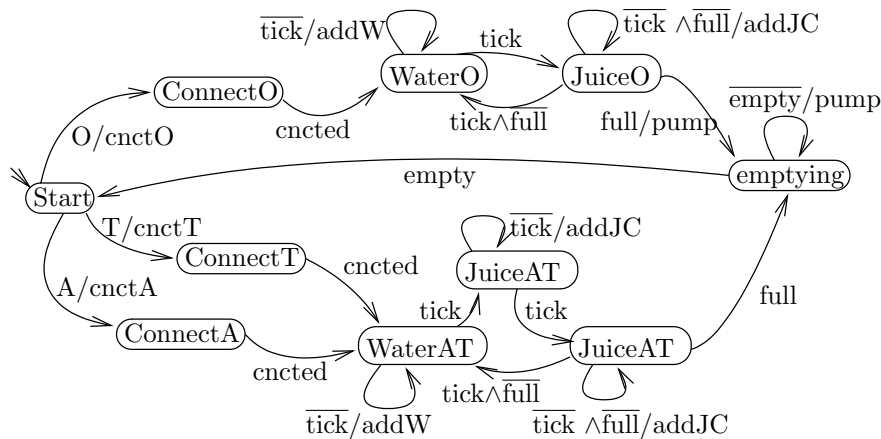


Fig. 10. A sample blender controller.

We then illustrate how to weave the aspect into this sampler blender controller; fig. 11 shows parts of the result. The set of *join points* selected by the pointcut program is reduced to the single **Emptying** state; the advice transitions are added from there.

The *recovery program* selects the points in the execution just after an occurrence of `cncted`. There is no state in the implementation (fig. 10) that corresponds exactly to this condition. For instance, we could think of the state `WaterO`, but the system can go from `Start` to `connectO` to `WaterO` to `JuiceO`

to **Water0** to **Juice0** ... to **emptying**. The first time it is in **Water0** matches the requirement of the recovery but not the other times.

The recovery states “just after an occurrence of **cncted**” are computed by the parallel composition of the controller (fig. 10) and the recovery program. This leads, for instance, to split the state **Water0** into two states, **Water0'** for the first time **Water0** is reached after the occurrence of **cncted** and **Water0''** for the other time (see fig. 11(a)).

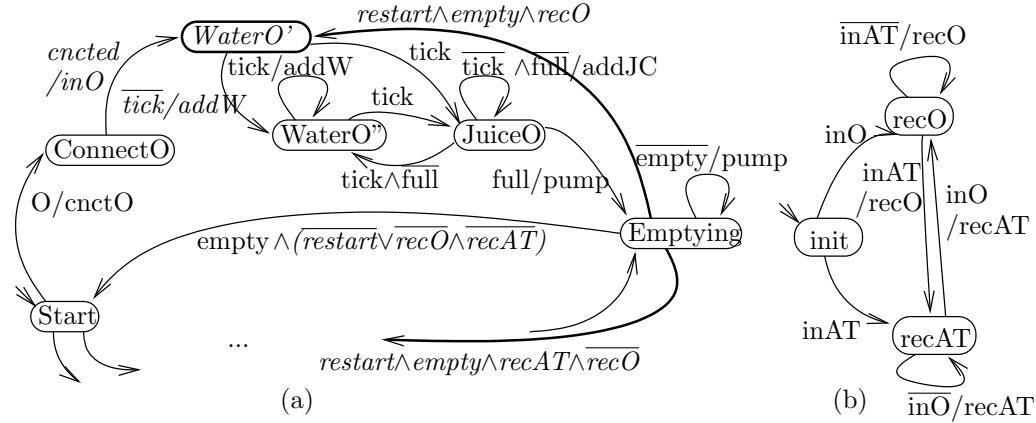


Fig. 11. Result of weaving the blender aspect in the controller. Added states, transitions and modified conditions are written in italic.

Once the join points and recovery states have been identified, transitions are added from the join point **Emptying** to the recovery states, e.g. **Water0'**. The upper half of the modified controller is displayed in fig. 11(a). However, the program must decide at runtime which of these transitions to take, because it has to go back to the last recovery state it has encountered. Thus, the controller must know which of the recovery states it passed last. This information is recorded in the *memory automaton*, which emits the information to the modified controller that is run in parallel.

The memory automaton for the blender controller is shown in fig. 11(b). It has three states: state **init** means that no recovery state has been passed so far, state **recO** means the last recovery state was **Water0'** and state **recAT** means the last recovery state was **WaterAT'**. The modified controller and the memory automaton communicate through four encapsulated signals. Each time the controller enters the recovery state **Water0'** (resp. **WaterAT'**), it emits **inO** (resp. **inAT**); this makes the memory automaton update its state by entering the state **inO** (resp. **inAT**). On the other hand, when the memory automaton is in its state **inO** (resp. **inAT**), it permanently emits the signal **recO** (resp. **recAT**). Consequently, when an advice transition is taken (the program is in the state **Emptying** and **restart** and **empty** are both true), the signal **recO** or **recAT** decides to which recovery state the controller goes back.

The triggering condition of the advice transition to the recovery state

`Water0'` is thus not only the activation signal `restart^empty`, but `restart^empty^rec0`, which indicates that `Water0'` was the last recovery state passed. A similar transition goes to `WaterAT'`, with condition `restart^empty^recAT^rec0`, where the additional `rec0` is needed to keep the automaton deterministic. The existing transitions of the join point are also reinforced by $\overline{\text{restart}} \vee \overline{\text{empty}} \vee (\overline{\text{rec0}} \wedge \overline{\text{recAT}})$. Thus, the automaton executes as before the aspect weaving when either `restart^empty` is false or no recovery state has been passed so far. The whole woven program controller is the parallel composition and encapsulation of the modified controller (see fig. 11(a)) and of the memory automaton (see fig. 11(b)).

4 Formal definitions

This section defines formally the concepts that were introduced informally in the previous sections.

4.1 The language

The language described in section 2 uses deterministic and complete automata, the parallel composition and the encapsulation. The *interface* of a program is made of its sets of inputs and outputs. For a program to be implementable, the following conditions should hold:

- the input and output sets are disjoint;
- it is *deterministic* (it behaves the same when executed with the same input sequence twice);
- it is *complete* (the behavior is defined for any input sequence).

4.1.1 Syntax

Definition 4.1 (*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, and $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{Bool}(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ is the set of transitions. $\mathcal{Bool}(\mathcal{I})$ 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, $\ell \in \mathcal{Bool}(\mathcal{I})$ is the triggering condition of the transition, and $O \subseteq \mathcal{O}$ is the set of outputs emitted whenever the transition is triggered.

Definition 4.2 (*Grammar of the language*). The set of expressions is defined by the grammar:

$$M ::= M \parallel M \quad | \quad M \setminus \Gamma \quad | \quad \mathcal{A} \quad (1)$$

where \mathcal{A} is an automaton (see def. 4.1), \parallel is the synchronous product, \setminus is the encapsulation and Γ is a set of Boolean variables.

The interface of an expression is given by the table in fig. 12, where \mathcal{A} is defined on the set of inputs (resp. outputs) $\mathcal{I}_{\mathcal{A}}$ (resp. $\mathcal{O}_{\mathcal{A}}$) and M_i is defined on inputs \mathcal{I}_i and outputs \mathcal{O}_i ($i = 1, 2$).

	in	out		in	out		in	out
\mathcal{A}	$\mathcal{I}_{\mathcal{A}}$	$\mathcal{O}_{\mathcal{A}}$	$M_1 \parallel M_2$	$\mathcal{I}_1 \cup \mathcal{I}_2$	$\mathcal{O}_1 \cup \mathcal{O}_2$	$M_1 \setminus \Gamma$	$\mathcal{I}_1 \setminus \Gamma$	$\mathcal{O}_1 \setminus \Gamma$

Fig. 12. The inputs and outputs of the composed programs.

Definition 4.3 (*Program*). A *program* on inputs \mathcal{I} and outputs \mathcal{O} is an expression given by the grammar (1), defined on \mathcal{I} and \mathcal{O} such that $\mathcal{I} \cap \mathcal{O} = \emptyset$.

4.1.2 Semantics

The semantics of a program P is given by defining the set of traces on its inputs and outputs which it may produce when executing. When P is a single automaton, we directly define its semantics as the set of all the possible traces of execution of the automaton; when it is a more complex expression, its semantics is that of the single automaton obtained when flattening the expression.

We first give some definitions about traces, then we define how to flatten an expression into a single automaton and finally, we derive the semantics of a program.

Sets of i/o-traces.

Definition 4.4 (*Traces*). Let \mathcal{I} , \mathcal{O} be sets of Boolean input and output variables representing signals from and to the environment. An *input trace* it is a function: $it : \mathbb{N} \rightarrow [\mathcal{I} \rightarrow \{\mathbf{true}, \mathbf{false}\}]$. An *output trace* ot is a function: $ot : \mathbb{N} \rightarrow [\mathcal{O} \rightarrow \{\mathbf{true}, \mathbf{false}\}]$. We denote by *InputTraces* (resp. *OutputTraces*) the set of all input (resp. output) traces. A pair (it, ot) of input and output traces (i/o-traces for short) provides the valuations of every input and output at each instant $n \in \mathbb{N}$. We denote by $it(n)[i]$ (resp. $ot(n)[o]$) the value of the input $i \in \mathcal{I}$ (resp. the output $o \in \mathcal{O}$) at the instant $n \in \mathbb{N}$.

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

Flattening an expression. Flattening transforms any expression into a single automaton. This is done by the function *flatten* which is inductively defined w.r.t. the grammar (1) as follows:

$$\text{Basis: } \textit{flatten}(\mathcal{A}) = \mathcal{A} \quad (2)$$

$$\text{Rule}(\parallel): \textit{flatten}(M_1 \parallel M_2) = \textit{flatProd}(\textit{flatten}(M_1), \textit{flatten}(M_2)) \quad (3)$$

$$\text{Rule}(\setminus): \textit{flatten}(M \setminus \Gamma) = \textit{flatEncaps}(\textit{flatten}(M), \Gamma) \quad (4)$$

where *flatProd* is the function that computes the synchronous product of two automata (def. 4.5) and *flatEncaps* is the function that encapsulates a set of variables Γ in an automaton (def. 4.6).

Definition 4.5 (*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 $\textit{flatten}(\mathcal{A}_1 \parallel \mathcal{A}_2) = \textit{flatProd}(\mathcal{A}_1, \mathcal{A}_2) = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{\text{init}1} s_{\text{init}2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ where \mathcal{T} is defined by:

$$\begin{aligned} (s_1, \ell_1, O_1, s'_1) \in \mathcal{T}_1 \wedge (s_2, \ell_2, O_2, s'_2) \in \mathcal{T}_2 &\iff \\ ((s_1, s_2), \ell_1 \wedge \ell_2, O_1 \cup O_2, (s'_1, s'_2)) \in \mathcal{T} . \end{aligned} \quad (5)$$

The synchronous product of automata is both commutative and associative.

Definition 4.6 (*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 $\textit{flatten}(\mathcal{A} \setminus \Gamma) = \textit{flatEncaps}(\mathcal{A}, \Gamma) = (\mathcal{Q}, s_{\text{init}}, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ where \mathcal{T}' is defined by:

$$\begin{aligned} (s, \ell', O', s') \in \mathcal{T}' &\iff (s, \ell, O, s') \in \mathcal{T} \wedge O' = O \setminus \Gamma \wedge \\ &\ell' \text{ is obtained from } \ell \text{ by setting all} \\ &\text{variables in } O \cap \Gamma \text{ to true, and all} \\ &\text{variables in } \Gamma \setminus O \text{ to false.} \end{aligned} \quad (6)$$

The encapsulation forces the synchronization according to Γ , as explained in section 2.2. Technically, if the triggering condition of a transition contains an encapsulated variable as a positive atom, and if the variable is not emitted by the transition, it is set to false by the encapsulation and the transition is thus cut. E.g., from the example in section 2.2, the transition $\bar{a} \wedge c0$ is cut, because the encapsulated variable $c0$ is set to false. Likewise, if the triggering condition of a transition contains an encapsulated variable as a negative atom, and if it

is emitted by the transition, the variable is set to true and the transition is also cut. E.g., transition $a \wedge \overline{c0} / c0$ is cut, because $c0$ is set to true and $\overline{c0}$ is thus false. In other cases, the transition is kept.

Definition 4.7 (*Semantics of an Automaton*). Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton. Its semantics is given in terms of a set of pairs of i/o-traces, $\text{Traces}(\mathcal{A})$. This set is built by using the following functions:

$$\begin{aligned} S_step_{\mathcal{A}} &: \mathcal{Q} \times \text{InputTraces} \times \mathbb{N} \longrightarrow \mathcal{Q} \\ O_step_{\mathcal{A}} &: \mathcal{Q} \times \text{InputTraces} \times \mathbb{N} \setminus \{0\} \longrightarrow 2^{\mathcal{O}} \end{aligned} \quad (7)$$

$S_step(s, it, n)$ 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 :

$$\begin{aligned} n = 0 &: S_step_{\mathcal{A}}(s, it, n) = s \\ n > 0 &: S_step_{\mathcal{A}}(s, it, n) = s' \quad O_step_{\mathcal{A}}(s, it, n) = O \\ &\text{where } \exists(S_step_{\mathcal{A}}(s, it, n-1), \ell, O, s') \in \mathcal{T} \\ &\wedge \ell \text{ has value true for } it(n-1). \end{aligned} \quad (8)$$

Let $it \in \text{InputTraces}$ and $ot \in \text{OutputTraces}$. We denote by $\text{Traces}(\mathcal{A})$ the set of all traces such that:

$$(it, ot) \in \text{Traces}(\mathcal{A}) \iff \forall n > 0. O_step(s_{\text{init}}, it, n) = ot(n-1). \quad (9)$$

This means that at each step n , the outputs emitted when executing the input trace it from the initial state are exactly the ones of ot .

Definition 4.8 (*Semantics of a program*). Let P be a program expressed as an expression. The semantics of P is given by its set of i/o-traces $\text{Traces}(P)$:

$$\text{Traces}(P) = \text{Traces}(\text{flatten}(P)) \quad (10)$$

Determinism and Completeness. the program P is said to be *deterministic* (resp. *complete*) iff its set of traces $\text{Traces}(P)$ is deterministic (resp. complete) (see def. 4.4). It is easy to show that the synchronous product preserves both determinism and completeness. In general, as explained in section 2.3, the encapsulation operation does not preserve determinism nor completeness. Programs are implementable if they are both complete and deterministic. In the sequel, we always consider programs with both properties.

Program equivalence. To conclude the formal definition of the language, we define a semantic equivalence of programs.

Definition 4.9 (*Semantic equivalence*). Let P and P' be two programs on the same sets of inputs and outputs. P and P' are said to be *semantically equivalent* (noted $P \sim P'$) iff $\text{Traces}(P) = \text{Traces}(P')$.

The equivalence of programs is defined on their sets of i/o-traces. It does not take the structure of the program into account, nor its states: two programs may be semantically equivalent without having the same expression, two automata may be semantically equivalent without having the same set of states or transitions.

This equivalence is a *congruence* w.r.t. the synchronous product and the encapsulation operator: applying one of those operators on programs preserves the semantic equivalence (see, for instance, [25]).

4.2 Larissa Aspects

The introductory examples in section 3 illustrate our notion of aspect and informally explain the weaving mechanism: roughly speaking, it consists in adding some transitions to the original program. This requires that an aspect specification contain the description of the source states, the target states and the labels of the added transitions. We call the added transitions the *advice* and their source states the *join points*.

As we want this program transformation to become a new operator in the language, we need it to preserve completeness, determinism and over all the semantic equivalence. Since the equivalence definition only relies on traces, we need to specify the target and source states of the added transitions in terms of traces.

4.2.1 Specifying pointcuts

This paragraph could have been entitled “how to specify states by a trace mechanism?”. The proposed solution is a mechanism that selects states according to the *history* of inputs, described by a synchronous *observer* P_{select} . Let P be a program on inputs \mathcal{I} and outputs \mathcal{O} . The input set of P_{select} is $\mathcal{I} \cup \mathcal{O}$ and its output set is a single fresh output $\{\text{SEL}\}$.

Selection of states: A state x of P will be considered as *selected* if, when P and P_{select} are run in parallel, SEL is emitted when a global state containing x is reached. If we simply put P and P_{select} in parallel, P 's outputs \mathcal{O} will become synchronization signals between them, as they are also inputs of P_{select} . They consequently will be encapsulated, and thus no longer emitted by the product. We avoid this problem by introducing a new output o' for each output o of P : o' will be used for the synchronization with P_{select} , and o will still be visible as an output. First, we transform P into P' and P_{select} into P'_{select} , 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:

$$\mathcal{P}(P, P_{\text{select}}) = (P' \parallel P'_{\text{select}} \parallel dupl_{o_1} \parallel \dots \parallel dupl_{o_n}) \setminus \{o'_1, \dots, o'_n\} \quad (11)$$

If $flatten(\mathcal{P}(P, P_{\text{select}})) = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ then the set of selected states is:

$$select(\mathcal{P}(P, P_{\text{select}}), \text{SEL}) = \{s' \mid (s, \ell, O, s') \in \mathcal{T} \wedge \text{SEL} \in O\} \quad (12)$$

Notice that the selection does not operate only on P , but on the synchronous product of P and some other automata. Due to P_{select} , the product may have more states than P alone: this occurs when the observer adds some information about the history of inputs that was not already computed by P . This case is illustrated by the blender recovery program in section 3.2.

From the pointcut program to join points: the join points of a program P are selected using the above mechanism. The observer is then called the pointcut program. If P is defined on inputs \mathcal{I} and outputs \mathcal{O} , the pointcut program P_{pc} is defined on inputs $\mathcal{I} \cup \mathcal{O}$ and output $\{JP\}$. The set of join points is given by $select(\mathcal{P}(P, P_{\text{pc}}), JP)$.

4.2.2 Specifying the advice

The advice is the set of transitions to be added. It contains two kinds of information: the labels of the added transitions (triggering condition, outputs emitted), and the target states (which state is reached after firing an advice transition). We define two kinds of advice, the *toInit advice* and the *Recovery advice*, which differ by the way they specify the target states. The first one, as in example 3.1, defines a single target state; it is the state that would be reached after executing a finite input trace from the initial state. The second one, as in example 3.2, selects recovery states of the program with a *recovery program* and the same selection mechanism as for join points. The advice transition then goes back to the last recovery state dynamically reached.

4.2.3 General aspect definition

Definition 4.10 (*Larissa aspect*). An aspect, for a program P on inputs \mathcal{I} and outputs \mathcal{O} , is a tuple $(P_{\text{pc}}, \text{type}, \text{advice})$ where

- $P_{\text{pc}} = (\mathcal{Q}_{\text{pc}}, s_{\text{pc}}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{\text{pc}})$ is the pointcut program.
- $\text{type} \in \{\text{toInit}, \text{Recovery}\}$ is the type of the advice.
- if “ $\text{type}=\text{toInit}$ ” then $\text{advice} = (\alpha, O, \sigma)$: α and O are respectively the triggering condition and the set of outputs emitted by the advice transitions. α is a Boolean expression over \mathcal{I} and fresh variables, and O is a set which may contain fresh variables as well as elements of \mathcal{O} . $\sigma : [0, \dots, \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow$

$\{\mathbf{true}, \mathbf{false}\}$ is a finite input trace of length $\ell_\sigma + 1$. It defines the single target state of the advice transitions by executing the trace from the initial state.

- if “*type=Recovery*” then *advice* = $(\alpha, O, P_{\text{rec}})$: α and O have the same meaning as for the *toInit* advice. $P_{\text{rec}} = (\mathcal{Q}_{\text{rec}}, s_{\text{rec},0}, \mathcal{I} \cup \mathcal{O}, \{REC\}, \mathcal{T}_{\text{rec}})$ is the recovery program: it selects the recovery states of P by emitting the output *REC*.

We define the semantics of aspects the same way we defined the semantics of programs, i.e. on a single automaton (sections 4.2.4 and 4.2.5). The weaving for programs is the natural extension:

Definition 4.11 (*Aspect weaving for programs*). Let P be a program and *asp* an aspect for P . $P \triangleleft \text{asp}$ is the program given by $\text{flatten}(P) \triangleleft \text{asp}$.

4.2.4 *toInit* advices

The weaving mechanism for a *toInit* advice consists in adding advice transitions from join points to the single state specified by the finite input trace σ ; join points are selected as shown in section 4.2.1. The pasteurizer example (section 3.1) dealt with an aspect with a *toInit* advice: it fully illustrates the effect of weaving.

Definition 4.12 (*Aspect weaving for automata – toInit advice*). Let $\mathcal{A} = (\mathcal{Q}, s_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and *asp* = $(P_{\text{pc}}, \text{toInit}, (\alpha, O_\alpha, \sigma))$ an aspect for a program on \mathcal{I} and \mathcal{O} . We note $\mathcal{A}_{P'} = (\mathcal{Q}_{P'}, s_{\text{init}P'}, \mathcal{I}_{P'}, \mathcal{O}_{P'}, \mathcal{T}_{P'})$ the automaton obtained when flattening the program $\mathcal{P}(\mathcal{A}, P_{\text{pc}})$ and $\mathcal{J} = \text{select}(\mathcal{P}(\mathcal{A}, P_{\text{pc}}), JP)$ is the set of join points (see eq. (11), (12) for the definition of \mathcal{P} and *select*). The weaving operator, \triangleleft , weaves *asp* on \mathcal{A} and returns the automaton: $\mathcal{A} \triangleleft \text{asp} = (\mathcal{Q}_{P'}, s_{\text{init}P'}, \mathcal{I}_{P'} \cup \text{var}(\alpha), \mathcal{O}_{P'} \cup O_\alpha, \mathcal{T}')$, where $\text{var}(\alpha)$ are the variables in α and \mathcal{T}' is defined as follows:

$$\left((s, \ell, O, s') \in \mathcal{T}_{P'} \wedge s \notin \mathcal{J} \right) \implies (s, \ell, O, s') \in \mathcal{T}' \quad (13)$$

$$\left((s, \ell, O, s') \in \mathcal{T}_{P'} \wedge s \in \mathcal{J} \right) \implies \left((s, \ell \wedge \bar{\alpha}, O, s') \in \mathcal{T}' \quad (14) \right.$$

$$\left. \wedge (s, \ell \wedge \alpha, O_\alpha, S_step_{\mathcal{A}_{P'}}(s_{\text{init}}, \sigma, l_\sigma)) \in \mathcal{T}' \right) \quad (15)$$

Transitions (13) are sourced in a non join point state: they are left unchanged. Transitions (14) are sourced in a join point state: to preserve determinism, their conditions are reinforced by $\bar{\alpha}$. Transitions (15) are the advice transitions, added by the aspect; their final state is specified by the finite input trace σ : $S_step_{\mathcal{A}_{P'}}$ (which has been naturally extended to finite input traces) executes the trace during l_σ steps, where l_σ is the length of σ , as defined in def. 4.10.

Reinitialization: a toInit example. An aspect that makes a program P reinitializable by \mathbf{r} is specified as follows: $reinit = (P_{pc}, toInit, (\mathbf{r}, \emptyset, \epsilon))$ where ϵ is the empty trace. P_{pc} is the automaton that chooses all states: $(\{q\}, q, \mathcal{I}, \{JP\}, \{(q, \mathbf{true}, JP, q)\})$. Weaving $reinit$ into P leads to adding a transition labelled by \mathbf{r} from any state of P to its initial state. Notice that $reinit$ can be applied to any program, regardless of its interface, since the condition “ \mathbf{true} ” can be interpreted as a condition on any set $\mathcal{I} \cup \mathcal{O}$.

4.2.5 Recovery advices

The weaving mechanism for a Recovery advice consists in adding advice transitions from join points to recovery states in such a way that the program goes back to the recovery state it passed last. A memory automaton remembers which recovery state was passed last. We motivated and introduced recovery aspects informally in section 3.2.

Definition 4.13 (*Aspect weaving for automata – Recovery advice*). Let $\mathcal{A} = (\mathcal{Q}, s_{init}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $asp = (P_{pc}, Recovery, (\alpha, O_\alpha, P_{rec}))$ an aspect for a program on \mathcal{I} and \mathcal{O} .

Join points and recovery states: we denote by $(\mathcal{Q}_{P'}, s_{init P'}, \mathcal{I}_{P'}, \mathcal{O}_{P'}, \mathcal{T}_{P'})$ the automaton obtained when flattening the program $P' = \mathcal{P}(\mathcal{P}(P, P_{pc}), P_{rec})$. P' is the program P in parallel with the pointcut program and the recovery program. The set of join points \mathcal{J} is equal to $select(P', JP)$ and the set of recovery states $\mathcal{R} = \{r_1, \dots, r_n\}$ to $select(P', REC)$.

Memory automaton: we define fresh variables $\mathcal{Rec} = \{rec_1, \dots, rec_n\}$ and $\mathcal{In} = \{in_1, \dots, in_n\}$: $in_i = \mathbf{true}$ means “the program enters the recovery state r_i (at this step)” and $rec_i = \mathbf{true}$ means “the last recovery state encountered is r_i ”. The memory automaton \mathcal{M} is given by $(\mathcal{Q}_{\mathcal{M}}, q_0, \mathcal{In}, \mathcal{Rec}, \mathcal{T}_{\mathcal{M}})$, where $\mathcal{Q}_{\mathcal{M}} = \mathcal{R} \cup \{q_0\}$ and $\mathcal{T}_{\mathcal{M}}$ is defined by

$$\begin{aligned}
& \forall i \leq n . \forall j \leq n . (r_i, in_j \wedge \bigwedge_{j < k \leq n} \overline{in_k}, \{rec_i\}, r_j) \\
& \forall i \leq n . (r_i, \bigwedge_{1 \leq j \leq n} \overline{in_j}, \{rec_i\}, r_i) \\
& \forall i \leq n . (q_0, in_i \wedge \bigwedge_{i < j \leq n} \overline{in_j}, \emptyset, r_i)
\end{aligned} \tag{16}$$

Advice and modified transitions: we note $P'_{\triangleleft} = (\mathcal{Q}_{P'}, s_{init P'}, \mathcal{I}_{P'} \cup \mathcal{Rec} \cup \text{var}(\alpha), \mathcal{O}_{P'} \cup \mathcal{In} \cup O_\alpha, \mathcal{T}_{P'_{\triangleleft}})$ where $\text{var}(\alpha)$ are the variables in α and $\mathcal{T}_{P'_{\triangleleft}}$

is defined by

$$s \in \mathcal{J} \quad \Longrightarrow \quad (s, \alpha \wedge \text{rec}_i \wedge \bigwedge_{j=i+1..n} \overline{\text{rec}_j}, O_\alpha, r_i) \in \mathcal{T}_{P'_{\triangleleft}} \quad (17)$$

$$\begin{aligned} (s, \ell, o, r_i) \in \mathcal{T}_{P'} \\ \wedge s \notin \mathcal{J} \end{aligned} \quad \Longrightarrow \quad (s, \ell, o \cup \{in_i\}, r_i) \in \mathcal{T}_{P'_{\triangleleft}} \quad (18)$$

$$\begin{aligned} (s, \ell, o, s') \in \mathcal{T}_{P'} \\ \wedge s \in \mathcal{J} \wedge s' \notin \mathcal{R} \end{aligned} \quad \Longrightarrow \quad (s, \ell \wedge (\overline{\alpha} \vee \alpha \wedge \bigwedge_{j=1..n} \overline{\text{rec}_j}), o, s') \in \mathcal{T}_{P'_{\triangleleft}} \quad (19)$$

$$\begin{aligned} (s, \ell, o, r_i) \in \mathcal{T}_{P'} \\ \wedge s \in \mathcal{J} \end{aligned} \quad \Longrightarrow \quad (s, \ell \wedge (\overline{\alpha} \vee \alpha \wedge \bigwedge_{j=1..n} \overline{\text{rec}_j}), o \cup \{in_i\}, r_i) \in \mathcal{T}_{P'_{\triangleleft}} \quad (20)$$

$$\begin{aligned} (s, \ell, o, s') \in \mathcal{T}_{P'} \\ \wedge s \notin \mathcal{J} \wedge s' \notin \mathcal{R} \end{aligned} \quad \Longrightarrow \quad (s, \ell, o, s') \in \mathcal{T}_{P'_{\triangleleft}} \quad (21)$$

Transitions (17) are the added advice transitions: they go from every join point to every recovery state. A transition to state r_i is taken when α is true and the memory automaton \mathcal{M} emits rec_i (meaning that the last recovery state is r_i). The equations (18), (19), (20) stand for the transitions of $\mathcal{T}_{P'}$ which have been modified and (21) for the unchanged transitions of $\mathcal{T}_{P'}$. Transitions (18) enter some recovery state r_i : they emit the output in_i to inform \mathcal{M} . Transitions (19) leave join points, which are no longer deterministic after the introduction of transitions (17). The expression $(\overline{\alpha} \vee \alpha \wedge \bigwedge_{j=1..n} \overline{\text{rec}_j})$ reinforces the conditions of transitions (19), such that the join points are again deterministic and complete. Finally, a transition (20) exits a join point to a recovery state r_i : it has thus both to inform \mathcal{M} by emitting in_i as in (18) and to reinforce its condition as in (19).

The woven program: finally, the woven program $\mathcal{A} \triangleleft \text{asp}$ is the synchronous product of P'_{\triangleleft} and \mathcal{M} being encapsulated by the variables in $\mathcal{R}ec$ and $\mathcal{I}n$:

$$\mathcal{A} \triangleleft \text{asp} = (P'_{\triangleleft} \parallel \mathcal{M}) \setminus (\mathcal{R}ec \cup \mathcal{I}n) \quad (22)$$

4.3 Extension of the language

Theorem 4.1 below expresses that weaving over an automaton preserves both completeness and determinism. Because of def. 4.11, so does the weaving over a program, whatever be its structure.

Theorem 4.1 (*Preservation of determinism and completeness*). Let P be a program and $\text{asp} = (P_{\text{pc}}, \text{type}, \text{advice})$ an aspect. Let P , P_{pc} and, if “type = Recovery” also P_{rec} , be deterministic and complete. Then, $P \triangleleft \text{asp}$ is also deterministic and complete.

Theorem 4.2 shows that the semantic equivalence is preserved when weaving an aspect.

Theorem 4.2 (*Preservation of equivalence*). Let P_1, P_2 be two programs on \mathcal{I} and \mathcal{O} and let asp be an aspect for a program on \mathcal{I} and \mathcal{O} . Then $P_1 \sim P_2 \implies (P_1 \triangleleft asp) \sim (P_2 \triangleleft asp)$.

Proofs for the theorems are given in appendixes A and B.

The completeness, the determinism and the equivalence being preserved, aspect weaving may be considered as a new operator in the language. Programs are now expressions built from single automata with the synchronous product, the encapsulation over a set of variables, and the weaving of an aspect. The new grammar for the language is now given by:

$$M ::= M \triangleleft asp \quad | \quad M \parallel M \quad | \quad M \setminus \Gamma \quad | \quad \mathcal{A} \quad (23)$$

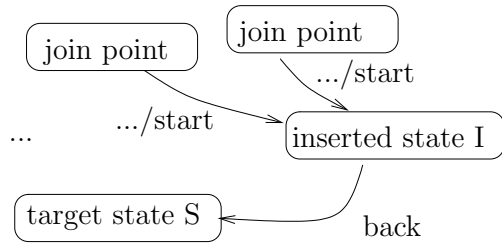
Example. As the weaving is a new operator of the language, it may be used in expressions such as: $((Blender \triangleleft restart) \parallel (Pasteurizer \triangleleft tester)) \triangleleft reinit$. *Blender* (resp. *Pasteurizer*) is the blender controller (3.2) (resp. the pasteurizer controller (3.1)), *restart* and *tester* are their respective aspects, and *reinit* is the reinitialization aspect (4.2.4). The above program is the global controller for the pasteurizer and the blender made reinitializable.

5 Implementation and extensions

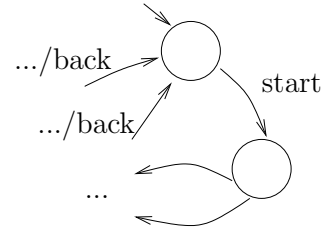
This section presents an implementation for Larissa, proposes an extension and briefly discusses aspect interference.

5.1 Implementation

A compiler [3] for Larissa was developed, as an extension of an existing Argos compiler. This tool is connected to simulation, test, debug and formal verification tools like model-checkers. We did some experiments with the model-checker Lesar [14]. We are currently developing examples. The current definitions are quite powerful (especially the recovery aspect), and seem to offer a good support for the kind of modifications we need in reactive programs.



(a) Transformation of the original program



(b) Advice automaton

Fig. 13. Inserting an advice program instead of advice transitions.

5.2 Extension: advice automaton

It may be useful to insert an entire program between the join point and the target state of the advice transition. First, note that introducing an automaton is enough since we can first flatten the program to be inserted. Then, the easiest way to insert an automaton is to give it separately and to synchronize it with the original program, in which it is sufficient to add a single state. The mechanism we propose is essentially the same construct as presented in the core of this paper, but for each target state S , a new inserted state I is added to the program. The advice transitions now go to I , and an additional transition goes from I to S . The transitions to I emit a special fresh output, **start** (begin the execution of the advice automaton), and the transition that leaves I is triggered by a single fresh input **back** (notifying the program that the advice automaton has finished). This is illustrated in fig. 13(a). The advice program (fig. 13(b)) has **back** as output and **start** as input, beside the normal inputs and outputs of the program. It is put in parallel with the program by the weaver, and **start** and **back** are encapsulated. The results and proofs of the paper can be easily extended for this kind of aspect, i.e., the weaving of an advice automaton preserves determinism, completeness and semantic equivalence: this kind of advice can also be considered as a new operator of the language.

5.3 Aspect Interferences

A key point when dealing with aspects is the notion of interferences. Generally speaking, if P is a program and \mathcal{A}_1 and \mathcal{A}_2 two aspects, the weaving of \mathcal{A}_1 and \mathcal{A}_2 into P may have different effects depending on how the weaver proceeds (weave \mathcal{A}_1 and then \mathcal{A}_2 ?, the reverse order?, weave \mathcal{A}_1 and \mathcal{A}_2 at the same time?). When this is the case, the aspects are said to interfere. Note that it may be the case that $(P \triangleleft \mathcal{A}_1) \triangleleft \mathcal{A}_2$ is well defined, while $(P \triangleleft \mathcal{A}_2) \triangleleft \mathcal{A}_1$ is not. In general, \mathcal{A}_2 is not an aspect for P alone – just look at the inputs and outputs – nor is \mathcal{A}_1 for $(P \triangleleft \mathcal{A}_2)$. Hence the interference problem cannot be expressed

simply as a comparison between $(P \triangleleft \mathcal{A}_1) \triangleleft \mathcal{A}_2$ and $(P \triangleleft \mathcal{A}_2) \triangleleft \mathcal{A}_1$.

In fact, from our point of view, the interference problem covers several situations leading to different questions. First situation, \mathcal{A}_1 may have been woven into P at some date; some properties φ hold for $(P \triangleleft \mathcal{A}_1)$. Later on, $(P \triangleleft \mathcal{A}_1)$ needs to evolve, by weaving \mathcal{A}_2 into it. The question of interference is then: does the weaving of \mathcal{A}_2 into $(P \triangleleft \mathcal{A}_1)$ break some effect of \mathcal{A}_1 , i.e. does φ still hold? Automated analysis tools (verification, testing, simulation...) can be used on $(P \triangleleft \mathcal{A}_1) \triangleleft \mathcal{A}_2$ and φ .

Second situation, imagine a modular program is designed as a single program P and two crosscutting features \mathcal{A}_1 and \mathcal{A}_2 . The program we want to obtain is P “plus” the two aspects woven in it. How to achieve this? The programmer meant to weave \mathcal{A}_1 and \mathcal{A}_2 “at the same time” into P . It seems that the weaving operator has to be extended to take into account not only a single aspect, but a set of aspects: $P \triangleleft \{\mathcal{A}_1, \mathcal{A}_2\}$. Intuitively, this operator is well-defined in simple cases, for instance when the set of join points for \mathcal{A}_1 and \mathcal{A}_2 are disjoint. For more complex cases, it is the responsibility of the programmer to specify the global modification, e.g. by another aspect $\mathcal{A}_{1,2}$.

From a practical point of view, in both situations, we can use Larissa’s connection to formal verification tools to check if some formally defined property φ still holds after the application of an aspect, and if \mathcal{A}_1 and \mathcal{A}_2 interfere in $P \triangleleft \{\mathcal{A}_1, \mathcal{A}_2\}$. However, aspects often invalidate properties and interfere even when one would not expect it. Furthermore, proving properties may be very expensive for large automata. We are currently investigating this issue to get a better insight into aspect interference at a higher level.

6 Related work

Comparison with AspectJ. Comparing Larissa with AspectJ is quite difficult, due to the difference of the underlying languages. Let us first compare AspectJ’s pointcuts with Larissa’s, and look at the blender example in section 3.2. To represent this pointcut in AspectJ, one would select the method that empties the tank as pointcut, and an `after` advice. This would cause the AspectJ-aspect to intervene at the same points as the blender aspect. However, this is possible only for very simple pointcuts. More complex pointcuts, which rely on the history of events, cannot be represented by a simple AspectJ pointcut: for instance, the pointcut of the pasteurizer aspect (section 3.1), must dynamically know whether the pasteurizer is switched on or off. Furthermore, in AspectJ it is impossible to jump to another point in the control flow as we do here. Instead, the aspect can execute program code, but after that, the execution continues at the position where the advice was

inserted.

On the other hand, it is not possible to directly express an arbitrary AspectJ `before`, `after` or `around` advice with the sorts of advice presented in this paper. Adding a kind of advice closer to AspectJ's advice could be done with two modifications of our advice. First, it is necessary to add general behaviors to the advice transitions, by adding an automaton between the join point and the final state. This idea is discussed in section 5.2. Second, the final state of the advice transition must not be globally fixed, but depend on the join point. Then, one could add the AspectJ advice as advice automaton (see fig. 13(b)), and let the target state be the join point. This inserts arbitrary code at join points and continues the execution afterwards, as `before` or `after` advice do.

Temporal pointcuts, trace-based aspects, stateful aspects, dynamic join points. A lot of authors have noticed before us that the `cflow` construct that enables dynamic join points in AspectJ is intrinsically limited by the nature of the history information contained in the runtime system (mainly the stack). This has motivated a lot of proposals for more general notions. In all these approaches, the idea is to trigger an aspect depending on some history, or “joint point sequence”, or “trace”, etc. In Arachne [8], aspects can be applied in sequences, which consist of restricted regular expressions over aspects, without parentheses or “or” constructs (ab^* is allowed, but not $a(bb)^*$, nor $a+b$). Stateful aspects, as defined in [7], may be the basis for the definition of automata-like pointcut languages, or general regular-expressions. In our approach, pointcuts are defined by finite automata, so we also have the power of regular languages. [28] is a temporal extension of AspectJ. Pointcuts can refer to method entries and exits, and one can construct expressions over these items with the power of context-free grammars. [21] proposes to define pointcuts with powerful predicates (à la Prolog) on traces.

Formal definitions of aspects. A number of approaches have been proposed for the formalization of aspects. Some of them aim at formalizing the notion of aspect itself, in the most general framework. Others are proposals for a notion of aspect in a given context, together with a formal definition, tailored for this precise context. Our proposal belongs to the second family.

Among the first family, we may cite [2]; the author proposes to define the weaving operation (for aspects of sequential programs) as a parallel composition; the formal framework is based on process algebras and products, and therefore very close to our setting. But we aim at defining aspects that *cross-cut* the parallel structure of reactive programs, not at formalizing weaving for sequential programs by a kind of parallel composition.

[29] is a formal definition of AspectJ-like advice, for a simple imperative language with recursive definitions. Our work does not easily compare to this

type of work, because the notion of aspect we chose for reactive systems is guided by the need to cross-cut a *parallel* structure. It is quite different from the AspectJ-like advice (“before”, “after” or “around” a piece of code) that make sense mainly for sequential languages.

Superimposition. Superimposition has been proposed by S. Katz and co-authors in several papers, starting with [17]. It can be considered as a precursor work on a notion of aspect for parallel programs. The authors proposed three categories of aspects called *spectative*, *regulative* and *invasive*, to qualify the effect of a superimposition on a program. Superimposition was mainly defined for distributed asynchronous systems on which one wants to impose a global property. It is however very close to our motivations. In our setting, a *spectative* superimposition is merely the synchronous composition with an *observer* [15], i.e. a component that observes the inputs and outputs of the original program, but cannot have a feedback effect on it. Invasive superimposition has the power to modify the behavior of a program drastically. To our opinion, it should therefore be defined in conjunction with strong properties relating its effect with the other constructs of the language. That is why we insist on being able to consider weaving as a new operator, obeying the same laws as the previous operators.

Enforcing properties of critical code. A number of approaches have been proposed for enforcing properties of programs, they mainly rely on dynamic checks.

In [6], a program transformation technique is presented, allowing to equip programs with runtime checks in a minimal way. Temporal properties are taken into account, and abstract interpretation techniques are used in order to avoid the runtime checks whenever the property can be proven correct, statically. In the general case, the technique relies on runtime checks, anyway.

The approach described in [26] is a bit different because it does not rely on program transformation. The authors propose the notion of *security automaton*. Such a security automaton is an observer for a safety property, that can be run in parallel with the program (performing an on-the-fly synchronous product). When the automaton reaches an error state, the program is stopped.

Edit automata [23] are more general. They allow a security specification to interfere with the program execution. An edit automaton may truncate the execution, suppress some actions, or insert some actions in the normal execution of a program. This technique is mainly dynamic and does not seem to be designed for program transformation, but we could probably *weave* such an edit automaton into an existing program by performing a kind of compiled synchronous product between them. The authors aim at executing unknown and untrusted programs in a safe way. This is different from our motivations,

because we would like the weaving operation to behave as a normal operation on programs. In particular, it should be possible to continue composing the woven program.

Aspects for automata-based languages. In [27], a notion of aspect is proposed for so-called “modular transition systems” (a kind of interpreted automata composed in parallel with shared variables). The ideas of the approach are quite similar to ours, including the proposal for formal automatic verification. The authors also show that their setting allows to give a clear definition to aspect interference. They do not study equivalences of programs, but propose to look at the properties that are preserved by the application of an aspect. The main differences with our work are: the focus on imperative code associated with transitions; the simpler structure of programs, which are sets of parallel automata. We concentrate on reactivity, and we use a general notion of program, with any level of operators.

A proposal for introducing aspects into Statecharts is made in [24]. Aspects are modeled as normal Statecharts, which are put in parallel with the base Statechart. Then, it can be specified that a transition in the aspect is taken before or after a certain transition in the base program. This approach does not have the semantical properties we are looking for, but has the advantage of being closer to AspectJ.

7 Conclusion and Further Work

This paper introduces a notion of aspects for a small language from the reactive synchronous languages family. The first motivation was that some recurring transformations on reactive synchronous programs appeared to be crosscutting notions.

Our main requirement on aspects is that the weaving transformation preserve the semantic equivalence between programs. We have defined and implemented two kinds of aspects: they both add new transitions in the program. The first one allows to go to some predefined state, when the aspect is activated. The second allows to go back to some recovery state. Their use is illustrated in a complete example. Finally, we propose some extensions of the weaving mechanisms and some comments on aspect interferences in this context.

The work leads to some interesting but more open questions. We examine two of them below.

Qualifying the effect of aspects A difficult to formulate question is *how much an aspect is allowed to modify a program?* or *how to qualify the relation-*

ship between the behavior of the program before and after the application of the aspect? S. Katz in [18] proposes to associate with a program a set of temporal logic properties that should be preserved by the application of any aspect. Our opinion is the following: first, it contradicts the obliviousness requirement; second, a lot of work on the specification and verification of reactive systems has proved that it is usually very hard to define a “complete” specification of a reactive system. We will study the opposite point of view: when defining an aspect, one should be able to provide a criterion for comparing the behaviors of programs before and after the application of this aspect.

Pragmatically, this can be done by studying the semantic effect of the aspect on each particular program. The criterion thus depends on the pair (program, aspect). Or it can only depend on the aspect: this could lead us to define some “semantic interface” of the aspect or some contract in the sense of aspect-aware interfaces [20] or crosscutting programming interfaces [12], that have been defined as tools to define the static effects of AspectJ aspects.

Could it be done without aspects? Wondering whether something we do with aspects could have been done “modularly” with the existing constructs of the language, is an interesting question, but not easy to specify. The most interesting setting to study this question is probably the notion of language expressivity proposed by M. Felleisen in [9]: a new construct C does add to the expressivity of a language if doing the same effect without it needs a lot of modifications everywhere in a program. The typical example he studies is the assignment construct, when added to a pure functional language.

References

- [1] K. Altisen, F. Maraninchi, D. Stauch, Exploring aspects in the context of reactive systems, in: Workshop on the Foundations of Aspect-Oriented Languages (FOAL), 2004.
- [2] J. H. Andrews, Process-algebraic foundations of aspect-oriented programming, in: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001), Vol. 2192 of LNCS, 2001, pp. 187–209.
- [3] Compiler for Argos and Larissa, <http://www-verimag.imag.fr/~stauch/ArgosCompiler/>.
- [4] A. Benveniste, G. Berry, Another look at real-time programming, Special Section of the Proceedings of the IEEE 79 (9).
- [5] G. Berry, G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, *Sci. Comput. Programming* 19 (2) (1992) 87–152.

- [6] T. Colcombet, P. Fradet, Enforcing trace properties by program transformation., in: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00), 2000, pp. 54–66.
- [7] R. Douence, P. Fradet, M. Südholt, Trace-based aspects, in: R. E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005, pp. 201–217.
- [8] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Ségura-Devillechaise, M. Südholt, An expressive aspect language for system applications with Arachne, in: Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, Chicago, IL, USA, 2005, pp. 27–38.
- [9] M. Felleisen, On the expressive power of programming languages, in: N. Jones (Ed.), ESOP'90 3rd European Symposium on Programming, Vol. 432 of LNCS, Springer-Verlag, 1990, pp. 134–151.
- [10] J. J. Fey, J. H. van Schuppen, VHS case study 4 - modeling and control of a juice processing plant, <http://www-verimag.imag.fr/VHS/CS4/dcs42.ps.gz> (1999).
- [11] R. E. Filman, D. P. Friedman, Aspect-oriented programming is quantification and obliviousness, in: R. E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005, pp. 21–35.
- [12] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, Modular software design with crosscutting interfaces, in: IEEE Software, Special Issue on Aspect-Oriented Programming, 2006.
- [13] N. Halbwachs, Synchronous programming of reactive systems, Kluwer Academic Pub., 1993.
- [14] N. Halbwachs, F. Lagnier, C. Ratel, Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE, IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems.
- [15] N. Halbwachs, F. Lagnier, P. Raymond, Synchronous observers and the verification of reactive systems, in: M. Nivat, C. Rattray, T. Rus, G. Scollo (Eds.), Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, 1993.
- [16] D. Harel, Statecharts: A visual formalism for complex systems, *Sci. Comput. Programming* 8 (3) (1987) 231–274.
- [17] S. Katz, A superimposition control construct for distributed systems, *ACM Trans. Prog. Lang. Syst.* 15 (2) (1993) 337–356.

- [18] S. Katz, Diagnosis of harmful aspects using regression verification, in: Workshop on Foundations of Aspect-Oriented Languages (FOAL'04), Lancaster, March 2004, 2004.
- [19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, LNCS 2072 (2001) 327–353.
- [20] G. Kiczales, M. Mezini, Aspect-oriented programming and modular reasoning, in: ICSE '05: Proceedings of the 27th international conference on software engineering, 2005, pp. 49–58.
- [21] K. Klose, K. Ostermann, Back to the future: Pointcuts as predicates over traces, in: Workshop on the Foundations of Aspect-Oriented Languages (FOAL), 2005.
- [22] L. Lamport, Proving the correctness of multiprocess programs, IEEE Trans. Softw. Eng. SE-3 (2) (1977) 125–143.
- [23] J. Ligatti, L. Bauer, D. Walker, Edit automata: Enforcement mechanisms for run-time security policies, International Journal of Information Security.
- [24] M. Mahoney, A. Bader, T. Elrad, O. Aldawud, Using aspects to abstract and modularize statecharts, in: 5th Aspect-Oriented Modeling Workshop, Lisbon, Portugal, 2004.
- [25] F. Maraninchi, Y. Rémond, Argos: an automaton-based synchronous language, Computer Language 27 (1/3) (2001) 61–92.
- [26] F. B. Schneider, Enforceable security policies, ACM Transactions on Information and System Security 3 (1) (2000) 30–50.
- [27] H. Sipma, A formal model for cross-cutting modular transition systems, in: Workshop on Foundations of Aspect-Oriented Languages (FOAL'03), 2003.
- [28] R. J. Walker, K. Viggers, Implementing protocols via declarative event patterns, in: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, ACM Press, New York, NY, USA, 2004, pp. 159–169.
- [29] M. Wand, G. Kiczales, C. Dutchyn, A semantics for advice and dynamic join points in aspect-oriented programming., ACM Trans. Prog. Lang. Syst. 26 (5) (2004) 890–910.

A Proof for Theorem 4.1

We prove that weaving preserves determinism and completeness. We prove the theorem for programs being simple automata, since weaving first applies the *flatten* function (see def. 2) on a program. Weaving an aspect is defined using parallel products, encapsulations and the adding of advice transitions. Parallel products preserves determinism and completeness. We will show that

so does the adding of advice transitions. Encapsulation does not always preserve determinism nor completeness: we will show that the particular cases in which encapsulation is used by the weaving preserve determinism and completeness. This is done by lemma A.1 – a proof is given at the end of this appendix:

Lemma A.1 Let $M_1 = (\mathcal{Q}_1, q_1, \mathcal{I} \cup A, \mathcal{O}_1 \cup B, \mathcal{T}_1)$ and $M_2 = (\mathcal{Q}_2, q_2, \mathcal{I} \cup B, \mathcal{O}_2 \cup A, \mathcal{T}_2)$ be two complete and deterministic automata such that $B \cap \mathcal{O}_2 = \emptyset$, $A \cap \mathcal{O}_1 = \emptyset$ and $A \cap B = \emptyset$. Furthermore, let all outgoing transitions of M_2 emit the same subset of A , i.e. $\forall t_1 = (s_1, \ell_1, o_1, s'_1), t_2 = (s_2, \ell_2, o_2, s'_2) \in \mathcal{T}_2$. $s_1 = s_2 \Rightarrow o_1 \cap A = o_2 \cap A$. (This means that the outputs of M_2 that are in A only depend on M_2 's current state, and not on the inputs and the taken transition.) Then, $M_1 \parallel M_2 \setminus (A \cup B)$ is complete and deterministic.

ToInit aspects. Let $asp = (P_{pc}, toInit, (\alpha, O_\alpha, \sigma))$. Let \mathcal{A} be an automaton. We prove that if \mathcal{A} and P_{pc} are deterministic and complete, then so is $\mathcal{A} \triangleleft asp$.

First, we proof that the computation of the join points preserves determinism and completeness: they are computed on the program $\mathcal{P}(\mathcal{A}, P_{pc}) = (\mathcal{A}' \parallel P'_{pc} \parallel dupl_{o_1} \parallel \dots \parallel dupl_{o_n}) \setminus \{o'_1, \dots, o'_n\}$. Calculating a parallel product does not affect determinism nor completeness. We then set $M_1 = \mathcal{A}'$, $M_2 = P'_{pc} \parallel dupl_{o_1} \parallel \dots \parallel dupl_{o_n}$, $A = \emptyset$ and $B = \{o'_1, \dots, o'_n\}$ to apply property A.1. Thus $\mathcal{P}(\mathcal{A}, P_{pc})$ is deterministic and complete.

Second, we show that adding advice transitions to $\mathcal{P}(\mathcal{A}, P_{pc})$ preserves determinism and completeness: some transitions are left unchanged, and any other with condition ℓ is replaced by two transitions with conditions $\ell \wedge \alpha$ and $\ell \wedge \bar{\alpha}$. This does not alter determinism nor completeness, thus $\mathcal{A} \triangleleft asp$ is deterministic and complete.

Recovery aspects. Let $asp = (P_{pc}, Recovery, (\alpha, O_\alpha, P_{rec}))$. Let \mathcal{A} be an automaton. We prove that if \mathcal{A} , P_{pc} and P_{rec} are deterministic and complete, then so is $\mathcal{A} \triangleleft asp$. The proof follows the same steps as above.

First, as for toInit aspects, $\mathcal{P}(\mathcal{A}, P_{pc})$ is deterministic and complete. The same reasoning is applied to show that $P' = \mathcal{P}(\mathcal{P}(\mathcal{A}, P_{pc}), P_{rec})$ is deterministic and complete.

Second, we show that adding advice transitions to $flatten(P')$ preserves determinism and completeness; this leads to P'_\triangleleft . Some transitions are left unchanged, and any other with condition ℓ is replaced by one transition with condition $\ell \wedge \bar{\alpha}$, one with $\ell \wedge \alpha \wedge \bigwedge_{1 \leq i \leq n} \overline{rec}_i$, and for every $i \leq n$ one with condition $\ell \wedge \alpha \wedge rec_i \wedge \bigwedge_{i < j \leq n} \overline{rec}_j$. These conditions are pairwise disjoint, the

automaton is thus deterministic. Their disjunction is ℓ , thus the automaton is complete. Hence P'_\triangleleft is deterministic and complete.

Third, the memory automaton, \mathcal{M} is constructed in such a way that it is deterministic and complete. Indeed, in any state r_i , we create n transitions with conditions $in_j \wedge \bigwedge_{j < k < n} \overline{in_k}$ and one transition with condition $\bigwedge_j \overline{in_j}$. These conditions are pairwise disjoint, the automaton is thus deterministic. Their disjunction is **true**, thus the automaton is complete.

Finally, $\mathcal{A} \triangleleft asp$ is obtained by $(P'_\triangleleft \parallel \mathcal{M}) \setminus (\mathcal{R}ec \cup \mathcal{I}n)$. Again, we apply property A.1 to show that the encapsulation does not affect determinism nor completeness: the memory automaton's output depend only on its state and not on the current input signals, so we can set $M_2 = \mathcal{M}$ and $M_1 = P'_\triangleleft$ and apply property A.1, with $A = \mathcal{R}ec$ and $B = \mathcal{I}n$. \square

Proof for lemma A.1 All outgoing transitions of a state s_2 in M_2 have the same subset of A in their outputs, say a . A complete monomial over a set of variables V is a Boolean conjunction which contains for each $v \in V$, either v or \overline{v} . We call \tilde{a} the complete monomial over A where the variables in a are positive and the variables not in a are negative. Furthermore, let \tilde{m} be a complete monomial over \mathcal{I} .

Because M_1 is complete and deterministic, a state s_1 of M_1 has exactly one outgoing transition $t_1 = (s_1, \ell_1, o_1, s'_1)$ with $\ell_1 \Rightarrow \tilde{m} \wedge \tilde{a}$. Let $b = o_1 \cap B$ and \tilde{b} the complete monomial over B where the variables in b are positive and the variables not in b are negative. Because M_2 is complete and deterministic, s_2 has exactly one outgoing transition $t_2 = (s_2, \ell_2, o_2, s'_2)$ with $\ell_2 \Rightarrow \tilde{m} \wedge \tilde{b}$.

In $M_1 \parallel M_2$, the combination of t_1 and t_2 leads to a transition $t = ((s_1, s_2), \ell_1 \wedge \ell_2, o_1 \cup o_2, (s'_1, s'_2))$. t isn't cut by the encapsulation and we obtain $t' = ((s_1, s_2), \exists(A \cup B) . \ell_1 \wedge \ell_2, (o_1 \cup o_2) \setminus (A \cup B), (s'_1, s'_2))$ in $M_1 \parallel M_2 \setminus (A \cup B)$. We thus have a transition for every complete monomial \tilde{m} in every state, the automaton is complete.

The automaton is also deterministic, because all the other transitions of the product are cut by the encapsulation. We show that for two states s_1 and s_2 and a complete monomial \tilde{m} , t' is the only transition to survive the encapsulation: Let $t'_1 = (s_1, \ell'_1, o'_1, s''_1)$ such that $t'_1 \neq t_1$ and $\ell'_1 \Rightarrow \tilde{m}$. Because $\ell'_1 \wedge \ell_1 = \mathbf{false}$, we have $\ell'_1 \wedge \tilde{a} = \mathbf{false}$, so there is variable $x \in a$ that either (1) appears as a positive atom in \tilde{a} and negated in ℓ'_1 or (2) the other way round. The subset of A in the outputs of all outgoing transitions of s_2 is a . Thus, after a product with a transition sourced in s_2 , the substitution of the encapsulation of A replaces variables in a by **true**, thus in (1), x is **false**. The substitution also replaces variables of A not in a by **false**, thus in (2), x is also

false. In both cases, the condition is false and the transition is cut. Likewise, let $t'_2 = (s_2, \ell'_2, o'_2, s''_2)$ such that $t'_2 \neq t_2$ and $\ell'_2 \Rightarrow \tilde{m}$. We have $\ell'_2 \wedge \tilde{b} = \mathbf{false}$ and the same reasoning applies, such that the transition is cut. \square

B Proof for Theorem 4.2

We prove the preservation of semantical equivalence for `toInit` aspects and recovery aspects. We prove the theorem for programs being simple automata, since weaving first applies the *flatten* function (see def. 2) on a program.

We first define the *equivalence between states*: let \mathcal{A}_1 and \mathcal{A}_2 be two automata; let q_1 be a state of \mathcal{A}_1 and q_2 a state of \mathcal{A}_2 , q_1 and q_2 are said to be equivalent (noted $q_1 \sim q_2$) iff $\mathcal{A}'_1 \sim \mathcal{A}'_2$ where \mathcal{A}'_1 (resp. \mathcal{A}'_2) is the automaton \mathcal{A}_1 (resp. \mathcal{A}_2) where the initial state is set to q_1 (resp. q_2).

ToInit aspects. Let $asp = (P_{pc}, toInit, (\alpha, O_\alpha, \sigma))$. Let \mathcal{A}_1 and \mathcal{A}_2 be automata. We prove that if \mathcal{A}_1 and \mathcal{A}_2 are semantically equivalent, then $\mathcal{A}_1 \triangleleft asp$ and $\mathcal{A}_2 \triangleleft asp$ are also semantically equivalent.

$\mathcal{A}_1 \sim \mathcal{A}_2 \implies \mathcal{P}(\mathcal{A}_1, P_{pc}) \sim \mathcal{P}(\mathcal{A}_2, P_{pc})$, because \mathcal{P} only relies on operators \parallel and \setminus , which are known to preserve equivalence. $\mathcal{P}(\mathcal{A}_1, P_{pc})$ and $\mathcal{P}(\mathcal{A}_2, P_{pc})$ consist of sets of equivalent states. The states in these sets are indistinguishable among themselves. Any set of equivalent states in an automata has an equivalent set of equivalent states in an equivalent automata. Thus, for $q_1 \in \mathcal{Q}_{\mathcal{P}(\mathcal{A}_1, P_{pc})}, q_2 \in \mathcal{Q}_{\mathcal{P}(\mathcal{A}_2, P_{pc})}$, we have $q_1 \sim q_2 \implies (q_1 \in select(\mathcal{A}_1, P_{pc}, JP) \iff q_2 \in select(\mathcal{A}_2, P_{pc}, JP))$; otherwise q_1 and q_2 would be distinguishable by JP .

If the aspect is applied to two equivalent states, they are modified in the same way. The input variables of the outgoing transitions are reinforced by α , the output variables by O_α . The added transitions also preserve equivalence, because the execution of σ from two equivalent states ends in equivalent states; otherwise the starting states would be distinguishable. Thus, $\mathcal{A}_1 \triangleleft asp \sim \mathcal{A}_2 \triangleleft asp$.

Recovery aspects. Let $asp = (P_{pc}, Recovery, (\alpha, O_\alpha, P_{rec}))$. We have already shown that $\mathcal{A}_1 \sim \mathcal{A}_2 \implies \mathcal{P}(\mathcal{A}_1, P_{pc}) \sim \mathcal{P}(\mathcal{A}_2, P_{pc})$. We obtain $\mathcal{P}(\mathcal{A}_1, P_{pc}) \sim \mathcal{P}(\mathcal{A}_2, P_{pc}) \implies \mathcal{P}(\mathcal{P}(\mathcal{A}_1, P_{pc}), P_{rec}) \sim \mathcal{P}(\mathcal{P}(\mathcal{A}_2, P_{pc}), P_{rec})$ by exactly the same reasoning. We denote $P'_i = \mathcal{P}(\mathcal{P}(\mathcal{A}_i, P_{pc}), P_{rec})$ for $i = 1 \dots 2$. $P'_{\triangleleft 1}$ and $P'_{\triangleleft 2}$ are not equivalent (they have even different in- and outputs: the In and Rec signals), nor are \mathcal{M}_1 and \mathcal{M}_2 . However, we

show that $(P'_{\triangleleft 1} \parallel \mathcal{M}_1) \setminus (\mathcal{R}ec_2 \cup \mathcal{I}n_2)$ and $(P'_{\triangleleft 2} \parallel \mathcal{M}_2) \setminus (\mathcal{R}ec_2 \cup \mathcal{I}n_2)$ are trace equivalent.

Let (it, ot) be a trace of $\mathcal{A}_1 \triangleleft asp$. By induction over the length of the trace, we show that (it, ot) is also a trace of $\mathcal{A}_2 \triangleleft asp$.

Inductive hypothesis:

$$\forall m \leq n . S_step_{\mathcal{A}_1 \triangleleft asp}(s_{init1}, it, m) \sim_{\bar{\alpha}} S_step_{\mathcal{A}_2 \triangleleft asp}(s_{init2}, it, m)$$

where the conditional equivalence $\sim_{\bar{\alpha}}$ is defined by

$$\begin{aligned} X \sim_{\bar{\alpha}} Y &\Leftrightarrow (\forall (it, ot) . (\forall n . (it(n) \Rightarrow \bar{\alpha})) \\ &\Rightarrow ((it, ot) \in Traces(X) \Leftrightarrow (it, ot) \in Traces(Y))) \end{aligned}$$

Base Case: For $n = 0$, S_step returns the initial state. We must thus show $\mathcal{A}_1 \triangleleft asp \sim_{\bar{\alpha}} \mathcal{A}_2 \triangleleft asp$. When α is false, only those transitions in $P'_{\triangleleft i}$ are taken that existed already in P'_i . The memory automaton has no effect on $P'_{\triangleleft i}$, because the transitions which take its outputs $\mathcal{R}ec$ into account also have α as condition, and are never taken. Thus, $\mathcal{A}_i \triangleleft asp \sim_{\bar{\alpha}} P'_i$ and because of $P'_1 \sim P'_2$, we have $\mathcal{A}_1 \triangleleft asp \sim_{\bar{\alpha}} \mathcal{A}_2 \triangleleft asp$.

Inductive step: We show that if the hypothesis is true for n , it is also true for $n+1$, and we also show $O_step_{\mathcal{A}_1 \triangleleft asp}(s_{init1}, it, n+1) = O_step_{\mathcal{A}_2 \triangleleft asp}(s_{init2}, it, n+1)$. We distinguish two cases: either (1) α is true, P_{pc} emits JP (i.e. we are in a join point) and P_{rec} has already emitted REC (i.e. we have already passed a recovery state), such that the aspect is activated, or (2) one of the above conditions is not met, and the aspect is not activated.

- (1) Both automata take advice transitions and emit O_α , we have thus $O_step_{\mathcal{A}_1 \triangleleft asp}(s_{init1}, it, n+1) = O_step_{\mathcal{A}_2 \triangleleft asp}(s_{init2}, it, n+1)$. Let r_1 and r_2 be the last recovery states passed in $1 \dots n-1$. They are passed at some instant $t_r < n$, the last instant in $1 \dots n-1$ where REC is emitted. In t_r , \mathcal{M}_1 (resp. \mathcal{M}_2) enters r_1 (resp. r_2), and emits rec_{r_1} (resp. rec_{r_2}) in n . Thus, $P'_{\triangleleft 1}$ (resp. $P'_{\triangleleft 2}$) takes the advice transition leading to r_1 (resp. r_2). Because of the induction hypothesis we have $r_1 \sim_{\bar{\alpha}} r_2$.
- (2) Only those transitions in $P'_{\triangleleft i}$ are taken that existed already in P'_i . Because of $P'_1 \sim P'_2$ and the induction hypothesis, we have $S_step_{\mathcal{A}_1 \triangleleft asp}(s_{init1}, it, n+1) \sim_{\bar{\alpha}} S_step_{\mathcal{A}_2 \triangleleft asp}(s_{init2}, it, n+1)$ and also $O_step_{\mathcal{A}_1 \triangleleft asp}(s_{init1}, it, n+1) = O_step_{\mathcal{A}_2 \triangleleft asp}(s_{init2}, it, n+1)$.

Because we inductively showed that $O_step_{\mathcal{A}_1 \triangleleft asp}(s_{init1}, it, n) = O_step_{\mathcal{A}_2 \triangleleft asp}(s_{init2}, it, n)$ holds for any n , $\mathcal{A}_1 \triangleleft asp$ and $\mathcal{A}_2 \triangleleft asp$ have the same outputs for it , thus (it, ot) is also a trace of $\mathcal{A}_2 \triangleleft asp$. \square