INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|___|___|___|___|___|___|___|___|___|___|

<u>T H È S E</u>

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité: "INFORMATIQUE: SYSTÈME ET LOGICIELS"

préparée au laboratoire VERIMAG

dans le cadre de l'École doctorale **"MATHEMATHIQUES, SCIENCE ET TECHNOLOGIES DE L'INFORMATION, INFORMATIQUE"**

présentée et soutenue publiquement

par

David STAUCH

le 13 novembre 2007

<u>Titre:</u>

Larissa, an Aspect-Oriented Language for Reactive Systems

Directrices de thèse:
Florence Maraninchi
Karine Altisen

JURY

Roland Groz	Président
Shmuel Katz	Rapporteur
Mario Südholt	Rapporteur
Pascal Fradet	Examinateur
Florence Maraninchi	Directrice de thèse
Karine Altisen	Directrice de thèse

Acknowledgements

First, I would like express my deepest gratitude to the members of the thesis committee. I would like to thank

Roland Groz, professor at INPG, for having accepted to preside the committee,

Shmuel Katz, associate professor at the Technion, Haifa, and *Mario Südholt*, maître de conférence at Ecole de Mines de Nantes, for the interest they have shown in my work, for the considerable energy they invested in reviewing it, and for the many interesting comments and suggestions in which this resulted,

Pascal Fradet, chargé de recherche at INRIA Rhône-Alpes, for having examined my Ph.D. as well as my master thesis, and for having guided my first steps towards understanding aspect-oriented programming,

Florence Maraninchi, professor at INPG, for having directed my work these last four years with discernment and kindness, and for having taught me those skills a Ph.D. student needs to survive, and

Karine Altisen, maître de conférence at INPG, for being a very accessible and helpful supervisor, and for her endless patience in discussing technical details and in proof-reading my thesis.

Next, I would like to thank the entire Verimag laboratory, and especially the "synchrone" team, for creating an agreeable, friendly and inspiring atmosphere in which my research could prosper and which made me nearly always go to the lab with pleasure. Of this atmosphere, an important part was provided by the fellow students with whom I shared my office, my lunch, and all kinds of interesting experiences. Without Aldric, Claude, Colas, FX, Giovanni, Guillaume, Hugo, Jacques, Jerome, Jonathan, Laure, Louis, Ludo, Manu, Mathias, Mouaiad, Olivier, Quentin, Simon, Sophie, and Tayeb, Verimag would not have been the same.

No praise is enough for the excellent administrative personnel at Verimag, who made administrative tasks a nearly pleasurable experience, nor for the great computer administration team, currently only composed of Jean-Noël, due to whose efforts I cannot remember a single computer related problem (well, in a strict sense) in my office.

This page would be missing something essential would it not express my thanks to all the friends I have made in the five years that I spent in Grenoble. Early (but lasting) acquaintances like Didier, JB, Luis, Ruben, Vass, Victor and Tob, the second year's bunch, including Amandine, Andrew, Antoine, Catherine, Cecile (L. and P.), Mario, Marion, Pascal, Thierry, and later arrivals like Anna, Elise, Emiliane, Helène, Julien, Laura, Matthieu, Marcus, Pierre, Sophie, and all that I forgot, will always be linked to my memories of Grenoble.

Finally, I would like to thank my parents, for everything, but especially for the “pot”.

Contents

1 Introduction **9**

2 Aspect-Oriented Programming and Reactive Systems **13**
2.1 Aspect-Oriented Programming 13
2.2 Reactive Systems 15
2.2.1 The Synchronous Approach 16
2.3 Adding Aspects to Reactive Systems 17
2.3.1 Criteria for a Synchronous Aspect Language 17
2.3.2 Choosing a Base Language 18

3 Argos **21**
3.1 Syntax and Intuitive Semantics 21
3.2 Formal Semantics 23
3.2.1 Traces and Trace Semantics 23
3.2.2 Automata 24
3.2.3 Argos Operators 26
3.3 Synchronous Observers 29
3.4 Some Examples of Encoding Aspects 30
3.5 Conclusion on Argos 31

4 Larissa **33**
4.1 Designing an Aspect Extension for Argos 33
4.2 A first Kind of Aspects: *toInit* and *toCurrent* 35
4.2.1 The Pasteurizer Controller 35
4.2.2 Variants of the Aspect Language 39
4.2.3 Formal Definition 40
4.3 Recovery Aspects 45
4.3.1 Example: the Blender 45
4.3.2 Formal Definition 49
4.4 Conclusion 52

5 Case Study: Modelling the Interface of a Complex Wristwatch 55
5.1 Introduction . . . 55
5.2 Modeling Interfaces of Small Devices . . . 56
5.3 Case Study: Suunto[1] Watches . . . 57
5.3.1 The Base Program . . . 57
5.3.2 The Altimax Watch . . . 57
5.3.3 The Vector Watch . . . 59
5.4 Related Work . . . 60
5.5 Conclusion . . . 61

6 Larissa Aspects as Argos Operators 63
6.1 Introduction . . . 63
6.2 Preservation of Determinism and Completeness . . . 64
6.2.1 Proof for Theorem 1 . . . 66
6.3 Preservation of Equivalence . . . 67
6.3.1 Proof for Theorem 2 . . . 67
6.4 Trace Transformation Semantics . . . 69
6.5 Conclusion . . . 71

7 Interference between Aspects 73
7.1 Introduction . . . 73
7.2 Example . . . 76
7.3 Interfering Aspects . . . 77
7.3.1 Extension to *recovery* Aspects . . . 78
7.3.2 Defining Interference . . . 79
7.4 Proving Non-Interference . . . 79
7.4.1 Interference between the Shortcut Aspects . . . 80
7.4.2 Interference between a Shortcut and the No-DTM Aspect . . . 81
7.5 Proofs . . . 82
7.5.1 Proof for Theorem 5 . . . 82
7.5.2 Proof for Theorem 4 . . . 86
7.6 Related Work . . . 86
7.7 Conclusion . . . 87

8 Contracts for Aspects 89
8.1 Introduction . . . 89
8.1.1 Synchronous Languages and Design-by-Contract . . . 89
8.1.2 Combining Contracts and Aspects . . . 90
8.2 Contracts for Argos . . . 91
8.3 Weaving Aspects in Contracts . . . 92
8.3.1 Formal Definitions . . . 92
8.3.2 Proof of Theorem 6 . . . 96
8.4 Example: The Tramway Door Controller . . . 99
8.4.1 Adding the Gangway . . . 100
8.4.2 Verification of the Woven Controller . . . 101
8.5 Related Work . . . 102

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

8.6 Conclusion 103

9 Implementation **105**
9.1 Introduction 105
9.2 Implementation of the Compiler 106
9.3 Syntax of Argos 106
9.3.1 A Simple Example 106
9.3.2 Parallel Product and Encapsulation 107
9.3.3 Process Calls 108
9.3.4 Refinement 108
9.3.5 Inhibition 109
9.4 Syntax of Larissa 109
9.4.1 Aspect Calls 109
9.4.2 toInit Aspects 109
9.4.3 Recovery Aspects 110
9.4.4 Inserting Advice Programs 110
9.5 Extensions 111
9.5.1 Integer Variables 111
9.5.2 Support for Contracts 113
9.5.3 File Inclusion 114
9.6 Towards Structure-Preserving Weaving 114
9.6.1 Structure-Preserving Weaving 115
9.6.2 Discussion 118

10 Related Work **121**
10.1 Aspects for Automata Languages 121
10.2 Stateful Pointcut Models 122
10.3 Formal Semantics and Properties for AOP 123
10.4 AOP and Modular Reasoning 125

11 Conclusion **127**
11.1 Context 127
11.2 Comments on the Contributions 128
11.2.1 Cross-cutting Concerns in Reactive Systems 128
11.2.2 Larissa, an Aspect Language for Argos 128
11.2.3 Semantic Analysis Tools for Larissa 129
11.2.4 Aspects for Parallel and Formally-Defined Languages 129
11.3 Perspectives 131
11.3.1 Extension to Integer Variables 131
11.3.2 Aspect Languages for Other Synchronous Languages 132
11.3.3 Non-Functional Concerns in Reactive Contexts 132
11.4 Concluding Remarks 134

Bibliography **135**

Chapter 1

Introduction

Software is used to solve more and more difficult problems since the invention of the computer. Thus, software systems increase in size, and are becoming harder to write, understand, and maintain. To master the inherent complexity, they are separated into modules, that fulfill different subtasks of the system. The way this decomposition is performed is decisive for the quality of software. Systems should be decomposed into modules in a way that is natural for the developer to think about them, and such that each module has a clearly defined task to solve. Ideally, each module should contain exactly one concern of the system, i.e. it should be responsible for implementing one functionality or concept. It should then be possible to modify the implementation of one concern by modifying only the module which implements it. This concept has been introduced by Parnas [Par72], and is generally termed "separation of concerns".

To perform the decomposition into modules, developers are constrained by the facilities that their programming language offers them. Numerous approaches propose different ways of creating modules, for example procedural, functional or object-oriented programming. These approaches have been very successful, and have allowed the construction of big systems that are clearly structured and easy to understand. However, the goal of isolating exactly one concern in a module cannot always be achieved with these approaches. Isolating a number of concerns into modules often means that other concerns cannot be placed in a module of their own, but have to be distributed over the existing modules. The code related to these concerns is then *scattered*, which means that it is distributed over several modules and not localized in a single one, and *tangled*, which means that it is mixed with the code of other concerns. This makes code difficult to write, understand, and maintain. Such concerns are called *cross-cutting concerns*, because they cut across the module structure.

Aspect-oriented programming (AOP) aims at isolating such cross-cutting concerns into a new kind of module, called an *aspect.* In AOP, non cross-cutting concerns are programmed in a base programming language, constituting the *base program.* Cross-cutting concerns are expressed as aspects in an aspect-oriented programming language, which extends the base language. The aspects are then *woven* (i.e. compiled) into the base program with an aspect weaver. This allows the expression of cross-cutting concerns in a module of their own, instead of being scattered and tangled into the base program.

Despite its relatively young age, AOP is being used widely. It has been particularly successful at encapsulating non-functional concerns in middleware for distributed enterprise applications written in Java, but it has also been applied to many other domains and programming languages. One field for which the use of aspect-oriented techniques has so far not been investigated, however, is the domain of safety-critical reactive systems. This thesis takes a first step in that direction.

The term *reactive systems* designates control systems which are in constant interaction with their environment, and which must respect real-time constraints imposed on them by their environment. They are common as embedded systems in the transportation domain, or as control systems for large industrial processes. They are very often safety critical, and must be verified with formal methods before they can be put into use. Therefore, they must be developed with programming languages with formal semantics. Furthermore, reactive systems are characterized by the need to perform different tasks in parallel. This should also be supported by the programming language.

Reactive systems are often developed in domain-specific programming languages, which support the use of formal methods, have a first class notion of parallelism, and are specially adapted to the input/output style of reactive systems. An example is the family of synchronous languages, which is widely used in industry. It is composed of different languages with different styles, including data-flow languages (Lustre [Hal93], Signal [LGLL91]) and imperative languages (Esterel [BG92], Argos [MR01]). All share a common semantics, centered around an explicit, high-level parallel composition of components.

Some cross-cutting concerns are also known to exist in these languages, e.g. the reinitialization of components. Due to the very different language constructs, they are usually quite different from those in general-purpose programming languages. To date, the concept of cross-cutting concern has never been explicitly studied in the context of synchronous programming languages.

This thesis investigates the notion of cross-cutting concerns and aspects in reactive systems written in synchronous languages. One first goal is to understand better what characterizes cross-cutting concerns in reactive systems. Therefore, we examine different examples of reactive systems and identify cross-cutting concerns. The examples we found are different from many common examples in Java in that they encapsulate *functional* concerns, i.e. concerns that model a part of the core functionality of the program. This suggests that cross-cutting non-functional concerns are less common in reactive systems. Our examples show that aspect-oriented programming is useful to encapsulate functional cross-cutting concerns.

Second, we consider how these cross-cutting concerns can be expressed with aspects. To this end, we have developed an aspect language for Argos, called *Larissa*. Argos has been chosen as base language because it is the simplest synchronous language. Its base element are Mealy automata, restricted to Boolean signals, which can be composed with a small number of operators. Unlike the other synchronous languages, Argos contains just the essence of synchrony. By developing an aspect language for it, we hope to get insight into aspect-oriented programming for synchronous languages in general.

To integrate well into Argos, aspect weaving should be considered another operator of the language, and an aspect a module. This means that the aspect weaving must respect the encapsulation of the module it advises, and only refer to its semantics, but not to its internal structure. Such formal properties ease understanding of programs both by programmers and formal analysis tools, and are strongly demanded by developers of reactive systems. In Argos, the respect of the encapsulation is formalized by requiring that operators must preserve semantic equivalence, i.e. applying an operator to two semantically equivalent programs yields two semantically equivalent programs. This ensures that operators can only refer to the semantics of the program, and not the way it is implemented. This differs from most other aspect languages, where aspects can refer to the internal structure of the programs they advise.

A clean definition of Larissa, which respects the encapsulation, has other advantages, too: it allows us to provide powerful tools to statically analyze programs written in Argos and Larissa, that are rarely available for aspect-oriented languages. We provide two such tools. One allows to analyze aspect interference precisely, and a second allows us to apply aspects not only to programs, but also to a specification of programs in the form of a contract.

Thus, the thesis claims that cross-cutting concerns exist in reactive systems, and that isolating these into aspects could benefit the development of reactive systems. It also discusses ways of specifying and implementing such aspects. It could hence be a first step towards the development of a full aspect language for reactive systems, which would be designed to be used in production environments.

The thesis is also interesting from an AOP point of view. It advocates the use of aspects with a firm semantic foundation, and illustrates the advantages of such an approach. Furthermore, the aspects in Larissa are placed in an uncommon setting, and are quite different from most other aspect languages. E.g., there are no methods in Argos, and thus the advice differs from the usual before/after/around advice which is predominant in most aspect languages.

Summary of Contributions

This thesis has thus the following contributions to make:

- The identification of examples of cross-cutting concerns in synchronous languages, most of them functional concerns. These examples are presented in Sections 4.2.1, 4.3.1, 5.3, and 8.4, and have been published in [AMS06a, AMS06b, Sta07b].

- The development of Larissa, an aspect language for the synchronous programming language Argos, including a formal definition, an implementation and the proof of important semantic properties. Larissa is presented in Chapter 4 and [AMS06a], the implementation is presented in Chapter 9 and available at [Lar], and formal properties, including the preservation of equivalence between programs, are proven in Chapter 6 and have been published in [AMS06a].

- Two semantic analysis tools for Larissa, namely a powerful interference analysis for Larissa aspects, and an algorithm to apply an aspect to a specification of programs in form of a contract. The interference analysis is presented in Chapter 7 and has been published in [SAM06] and [Sta07a], and the contract weaving is presented in Chapter 8 and published in [Sta07b] and [Sta07a].

- A discussion of specific requirements formally-defined and parallel languages have towards aspect-oriented languages. It takes place in Section 2.3 and in Section 11.2.4 in the conclusion.

Outline of the Document

The remainder of the document is structured as follows. Chapter 2 explains the context of this thesis, i.e. aspect-oriented programming and reactive systems. We also discuss requirements for aspect-oriented extensions for synchronous languages, and further motivate the choice of Argos as a base language. Chapter 3 presents Argos, the base language for Larissa. Chapter 4 introduces the different kind of aspects in Larissa, and explains them with a typical example of a reactive system. Larissa is also defined formally. Chapter 5 presents a case study performed using Argos and Larissa, the modeling of the interface component of a complex wristwatch. Larissa is used to modularly express cross-cutting concerns, and to build a product-line of several watches. Chapter 6 postulates that aspect weaving can be considered as a new operator, by proving some important properties, namely the preservation of determinism and completeness of programs, and the preservation of equivalence between programs.

The next two chapters make use of the semantic properties of Larissa, and offer powerful analysis tools for aspects. Chapter 7 studies interference between aspects, and allows to prove non-interference in certain cases. Two aspects can be shown not to interfere, either independently of the program they are applied to, or for a specific program. This analysis is performed statically and at low cost. Chapter 8 proposes a way to apply an aspect to a class of programs specified by a contract, instead of applying it to a specific program. This yields a new contract, which holds for any program which fulfilled the original contract, with the aspect applied to it. An example illustrates how this mechanism can be used for modular verification.

Chapter 9 explains the implementation for Argos and Larissa, and Chapter 10 exposes related work for Larissa in general. Related work specific to one of the earlier chapters is discussed there. Chapter 11 concludes the thesis, and gives some perspectives.

Chapter 2

Aspect-Oriented Programming and Reactive Systems

2.1 Aspect-Oriented Programming

A recurrent problem in software development are cross-cutting concerns, which are concerns that cannot be encapsulated into modules using standard decompositions. Aspect-oriented programming aims at the encapsulation of cross-cutting concerns into aspects, which can then be woven into the program. It is a relatively new programming paradigm, having been introduced over the last ten years. Whereas the first aspect-oriented languages were tailored for specific problems [KLM+97], such as performance optimisation or synchronization, general-purpose aspect languages were soon developed, most prominently among them AspectJ [KHH+01, Asp]. These languages offer a number of constructs to the programmer, the most important being the possibility to attach advice to join points, which are selected by a pointcut.

Join points are well-defined points in the execution of the program, e.g. a call to a method `foo`, or the writing or reading of a variable `v`. A *pointcut* is an abstract description of a set of join points in a program execution. Examples are "all calls of method `foo`", "all calls to methods whose name begins with `set`", or "all calls to method `foo` inside method `bar`". A pointcut thus selects certain join points during the execution of a program. Finally, a piece of *advice* is executed at the join points selected by the pointcut. The advice can be any piece of code, with access to the context of the intercepted join point. E.g., advice can change the parameters of method `foo`, or prevent its execution. Thus, an aspect is composed of a pointcut and a piece of advice, and it modifies the execution of a program by executing the advice at join points selected by the pointcut.

There are aspect-oriented extensions to many different languages, but the most widely used is AspectJ, an extension to Java. Consider the simple example in Figure 2.1, which has been taken from the compiler for Argos and Larissa presented in Chapter 9. This aspect measures the time taken by the different passes of the compiler. Each pass is performed by a public method of the class `argos.ArgosCompiler`. Therefore, we first write a pointcut that selects all calls to these methods (lines 6 and 7). It consists of the keyword `pointcut`,

```
1  public aspect Profiling {
2
3    private String message = "";
4    private long start=0, sum=0;
5
6    pointcut pass() :
7      call(public * argos.ArgosCompiler.*(..));
8
9    before(): pass() {
10     start = System.currentTimeMillis();
11   }
12
13   after() : pass(){
14     long duration = System.currentTimeMillis() - start;
15     sum += duration;
16     message = message + thisJoinPoint.getSignature()
17               + "␣took␣" + duration + "ms";
18   }
19   ...
20 }
```

Figure 2.1: Example of an AspectJ aspect.

its name `pass`, the keyword `call`, that signifies that we are selecting method calls, and the signature of the methods we are selecting, here all public methods in `argos.ArgosCompiler`, with any return type, name and parameters. Arbitrary types or names are specified with a `*`, and an arbitrary parameter list is specified by `..`.

AspectJ offers three kinds of advice: `before`, `after`, and `around` advice, which are respectively executed before, after, or instead of an intercepted event. In the example, we use a `before` advice (line 8) to remember the time at which the method was called, and an `after` advice (line 12) to calculate the time it took. An advice starts with the keyword denoting its type, followed by the pointcut it advises, and the Java code that is to be executed at the selected join points. Pointcuts and advice are included into an aspect, which is a module similar to a Java class. Besides pointcuts and advice, it can contain methods and variables (e.g. lines 3–4 in Figure 2.1).

AspectJ is much richer than the part that is illustrated by the example above. Pointcuts can select join points using many different criteria, including the current call stack, the textual locality of the code, and the dynamic type of the executing object. Advice can also access the context of join points, e.g. the calling object. Aspects can also change the static type structure of programs: methods and variables can be declared in an aspect, and then introduced into a class or an interface at compile time. Furthermore, aspects can also modify the type hierarchy of a class, by making it implement another interface or extend another class.

Although AspectJ is the most successful and mature aspect language, aspect extension for many different languages exists, e.g. AspectC++ [SGSP02] for C++, AspectC [CK03] for C, Eos [RS03] for C#, or extensions to functional languages like Aspectual Caml [MTY05] for Caml or PolyAML [DWWW05] for ML. These aspect languages share these features to a greater or lesser extent, and they introduce new features, which are adapted to their base language or specific needs. However, the pointcut/advice model is common to all aspect languages, and all pointcut languages can select method calls or executions.

Aspect-oriented programming is being used to modularize different cross-cutting concerns in very different applications. It has met particular success in middleware for distributed enterprise applications. In such systems, aspects are very successfully used to modularize non-functional concerns. Non-functional concerns implement a functionality that is not part of core- or business logic of the program, and tend to be cross-cutting. Common examples include transaction management and data persistence (e.g. [KG06, RC03]), synchronization and distribution (e.g. [NSV$^+$06, SLB02]), exception handling (e.g. [MG07]), caching (e.g. [SDMML03]), and security (e.g. [DW06, VBC01]). Aspects that help the development process are also very popular, e.g. tracing, logging, profiling, or the enforcement of programming rules (e.g. [CC04]).

However, there are also many functional concerns that are cross-cutting and that can be modeled with aspects. A classical toy example is the updateDisplay aspect [KHH$^+$01], which updates a display whenever the state of the displayed object has changed. Aspects are also used to build product lines (e.g. in [LHB05]), i.e. to add additional functionality to a product to build different versions. Besides middleware, AOP has been applied to many other domain, e.g. operating systems [CK03].

2.2 Reactive Systems

Reactive systems are systems that are in constant interaction with their environment. They constantly receive inputs from their environment, and emit outputs to it, thus producing an infinite input/output trace. They have been distinguished by [HP85] from transformational systems (e.g. compilers), that calculate a result from some initial data. They can be further separated from interactive systems like user interfaces or operating systems, which are also in constant interaction with their environment, but can impose their own rhythm, by letting the environment wait until they finished their computations. Reactive systems, on the other hand, must react to inputs from the environment within a given time, which is imposed by the environment.

Typical examples of reactive systems are control systems in the transportation domain (e.g. in airplanes, trains, and automobiles) or for large industrial systems (e.g. power plants). They are thus very often highly safety critical, and erroneous behavior of the software may have fatal consequences. Furthermore, once a system is in use, it is rarely possible to update the software. For these reasons, formal methods are usually used in the development process, for the specification, verification, or testing of the software. This in turn requires programming languages with formal semantics. Usually, a suitable model of the program is verified, with techniques such as model checking, abstract interpretation or theorem proving. Such a model can be derived from the code directly, or from a specification from which the code is generated.

Reactive systems are further characterized by the need to fulfill several tasks in parallel. E.g., a safety system controller in a car must survey many sensors, measuring different parameters of the automobile, and take different actions, e.g. switch a warning light on or activate the airbag. Some of these tasks are independent, others need to interact. Therefore, programming languages for reactive systems must offer explicit support for the parallel combination of program modules, and the communication between them.

Furthermore, reactive systems are usually required to be complete. This means that the program should emit a sequence of outputs for every possible sequence of inputs. This is expected of a reactive systems which should run forever, without any blocking. A second important property is determinism, which requires that a program always emits the same sequence of outputs for a sequence of inputs. Determinism makes testing and verifications of programs easier, but cannot always be guaranteed, e.g. in distributed systems.

2.2.1 The Synchronous Approach

Most reactive systems are not distributed, but executed on a single processor. However, they often execute several tasks in parallel, and are most naturally described as parallel units, and not as a monolithic program. This is true for most computer programs, and is generally addressed by executing multiple threads on the same processor. However, dynamically scheduling threads makes programs non-deterministic and verification much more difficult. Synchronous languages offer a different solution to this problem. They describe programs as parallel components, but compile them into sequential code. The parallel composition of components has a clear semantics, and programs are checked at compile time for determinism and completeness. Because the parallel composition is deterministic, the global state space of the program is reduced, and verification becomes much easier.

The semantics of synchronous languages is based on the *synchrony hypothesis*. This hypothesis divides time into instants, and assumes that reactions are atomic, i.e. that in each instant, each component instantly updates its outputs following the new inputs. Components composed in parallel communicate with the *synchronous broadcast*, meaning that each component can read the outputs of the other components in the instant as inputs. This allows an easy and elegant way of communication, because emitting components do not have to know who is reading their outputs, and need not explicitly send messages to other components.

Finally, all synchronous programs can be compiled into a simple program of the following form:

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

Of course, such a compiled program emits the outputs some time after it received the inputs, and thus is not truly synchronous. However, if this time is smaller than the minimal reaction time of the environment of the system, it can be considered as such. Therefore, we must be able to give an upper bound on the cycle time. Before a program is put into an environment, it is verified that the cycle time of the program is smaller than the reaction time of the environment. Because they must give an upper bound of the cycle time, synchronous languages do not contain any constructs that would make the calculation of the worst case execution time difficult, such as recursion or unbounded loops.

Starting in the eighties, several synchronous languages have been developed in France, namely Lustre [Hal93], Esterel [BG92], Signal [LGLL91], and Argos [MR01]. Lustre and

Signal are data-flow languages, and share a declarative style. Each variable is considered to be a *flow*, with a value at each instant. Programs are structured into data-flow networks, and the outputs of each node in the network are defined by a system of equations. Not all nodes must be active at each instant, but they can be activated individually by *clocks*. Lustre and Signal differ in the treatment of clocks attached to signals. Whereas each Lustre program has a base clock, and all other clocks are slower than this base clock, Signal can have several independent clocks. Lustre is commercialized in the Scade tool [Sca] by Esterel Technologies, which offers a graphical description of Lustre. Because Scade includes a code generator certified for safety-critical aeronautical applications, it is being used successfully to develop such systems. E.g., much of the safety-critical software of the Airbus A380 is developed using Scade. Signal, on the other hand, is commercialized in the Sildex tool, which is offered by TNI Software [TNI].

Esterel is an imperative language, which is particularly suited for describing control structures. It has been used in the aviation industry, and is also used to synthesize circuits. It is industrialized by Esterel Technologies in Esterel Studio Suite [Est].

Unlike the other synchronous languages presented here, Argos is not a production language and is not used in an industrial context. It is a small and elegant language, which contains the central constructs of synchronous languages, like the parallel composition. Argos is an automata language, its basic elements are Mealy automata, which can be combined with a number of operators. It is similar to Statecharts [Har87], although it does not include all of Statecharts' features, and it has a well-defined synchronous semantics.

2.3 Adding Aspects to Reactive Systems

2.3.1 Criteria for a Synchronous Aspect Language

Some cross-cutting concerns are known to exist in reactive systems written in synchronous languages. E.g., it is common that a system should be re-initializable, i.e. one should be able to put the system back in its initial state with a special input "reset". This can be done modularly in Esterel or Argos, but needs a modification of every line of code in Lustre. It is thus a typical cross-cutting concern in this language. In Esterel, some specific needs such as re-initialization have led to the introduction of dedicated constructs. Such dedicated constructs improve the language as long as their number is limited. Introducing too many of them, however, makes code difficult to understand. Furthermore, each time a new cross-cutting concerns appears, a new construct must be introduced. Introducing a generic aspect-oriented language, on the other hand, may express a large variety of cross-cutting concerns with a small number of new constructs.

These cross-cutting concerns suggest that synchronous languages could profit from aspect-oriented programming. It seems therefore justified to investigate the combination further, to support or refute this claim. Therefore, we must first develop an aspect-oriented extension to a synchronous language. One option would be the adaption of an existing aspect language like AspectJ to a synchronous language, thus introducing pointcuts which intercept functions calls based on method names, and `before`, `after`, and `around` advice. However, this approach has several disadvantages.

A first problem stems from the fact that synchronous languages are *parallel* languages, whereas the common concepts of aspect languages apply to *sequential* language. Being parallel languages, synchronous languages are structured into parallel modules, such as nodes in Lustre, in the same way that object-oriented languages are structured into classes and methods, or functional programming languages into packages and functions. This parallel structure is the main decomposition of the synchronous languages, and any aspect must thus naturally cut across it. Unlike methods or functions, these parallel units are each executed at each instant, in parallel. `before`, `after`, and `around` thus loose their meaning.

We can try another comparison between parallel and sequential languages. Instead of equating the parallel modules to the structuring units in sequential language, we can take the point of view that aspects have interface elements as join points in sequential languages, and could also do so in parallel languages. Thus, we could take inputs and outputs as join points, and then insert traditional advice, e.g. we could emit another signal after a certain in- or output becomes true. However, this can already be done with parallel modules. It is easy to modularly intercept input or output signals of a synchronous program with the parallel composition. The signal can then be delayed, suppressed or replaced by other signals, something that would require an aspect in other programming languages. Thus, as has also been noted by Bruns et al [BJJR04, 219], the parallel composition in synchronous languages already includes a part of the power of aspects in sequential languages. Indeed, Andrews [And01] uses a variant of Process Algebra [Hoa85, Mil80], a formalism similar to synchronous languages, to encode an AspectJ-like aspect extension of a simple imperative language. We show in Section 3.4 how some typical concerns that are often expressed with aspects in sequential languages can be expressed with the existing Argos operators.

A second problem arises because synchronous languages are formally-defined, and are used to develop safety-critical applications. This imposes a number of restrictions on all language extensions, including aspect languages. Thus, aspects should have a clear and simple semantics, and must not disrupt the connection between the programs and the formal analysis tools. This can be achieved by a definition of weaving that produces code in a format that the tools can read, preferably the base language. In addition, aspects for formally-defined languages should maintain the semantic properties of the language, which guarantee programs to be correct-by-construction. In synchronous languages, aspects should especially preserve determinism and completeness. Furthermore, modules in synchronous languages are strongly encapsulated, only their in- and outputs are visible from the outside, but not their inner structure. Aspects should also respect the encapsulation of their base program. They must hence not refer to the inner structure of the program they are applied to, such as the names of processes or local variables. This is different from most other aspect languages, which can refer to invisible elements such as private methods.

Thus, although an aspect-oriented extension for a synchronous language must be inspired by existing aspect languages, it cannot directly adopt all the common concepts that they share. It must instead change these concepts to adapt to the specific requirements of the host language, and to be able to express recurrent cross-cutting concerns.

2.3.2 Choosing a Base Language

An important choice to make is the base language. Given the very different styles of synchronous languages, aspect languages for the different synchronous languages are likely to look quite different. However, there are also commonalities among them, most notably the parallel composition, which is the main structuring element in all the languages. Thus, we hope that developing an aspect extension for one language will provide insight into the nature of aspect languages for synchronous languages in general.

Extending Lustre, Esterel, or Signal would have the advantage to be working with a real production language, with many real-world examples and potential users available. However, these languages are rather complicated, and extending them would be a considerable effort, such that playing with different alternatives would become difficult. Furthermore, any extension would likely solve specific problems of the chosen language, instead of exploring aspect orientation to synchronous languages in general.

Thus, given that there is no prior work on aspects in synchronous languages, a simple language is more appropriate for a first experimentation. Argos seems a good candidate. Indeed, a module of a synchronous language consists in its simplest form of its interface, a representation of its internal state, and a function relating the outputs to the internal state and the inputs. The basic modules of Argos are Mealy automata and thus correspond exaclty to this description. Furthermore, it should be possible to compose modules in parallel, and to let them communicate with the synchronous broadcast. Argos contains little more than this, and can thus be situated not far from the simplest synchronous language possible. Using Argos as base language has the following advantages:

- it has a simple semantics that can be understood and extended with little difficulty; it may also be possible to obtain interesting semantic properties for the aspect language.
- the language is small and thus simple to implement, allowing easy experimentation with different features and approaches.
- it is nonetheless expressive enough to write meaningful examples, and has all the characterizing elements of a synchronous language, thus being a viable representative of the other synchronous languages.

For these reasons, we chose Argos as base language for a synchronous aspect language. The next chapter presents Argos in more detail.

Chapter 3

Argos

This chapter defines Argos, the base language for the aspect language introduced in the next chapter. We first introduce the graphical syntax and the intuitive semantics of Argos, and then give formal definitions. Argos has been first published in [Mar91], and is discussed in detail in [MR01].

3.1 Syntax and Intuitive Semantics

We first describe the basic element of Argos, flat Mealy automata, and then introduce the four existing operators. The first two, the parallel product and the encapsulation, are central to the definition of aspects, and the remaining two will be used in example programs.

Figure 3.1 shows a simple automaton with input `a` and output `b` which emits a `b` every two `a`'s. Rounded-corner boxes are automaton states, in the example states `0` and `1`. State `0` is the initial state, designated by an arrow without a source state. States are connected by transitions, displayed as arrows. A set of states and transitions which are connected together constitutes an automaton. Transitions are labeled by a Boolean condition on input signals, and a set of emitted signals. We use the concrete syntax `condition / emitted signals`. In the condition, negation is denoted by overlining and conjunction is denoted by a dot. When the output set is empty, it can be omitted. Examples are `a/b` or $\texttt{a}.\overline{\texttt{c}}$. States are named, but names should be considered as comments: they cannot be referred to in other components nor are they used to define the semantics of the program. An arrow can have several labels — and stand for several transitions, in which case the labels are separated by a comma. By convention, every automaton is 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. E.g., the automaton in Figure 3.1 is supposed to have

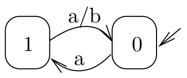


Figure 3.1: A simple automaton, which emits a `b` every two `a`'s.

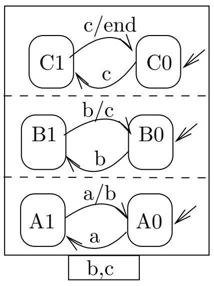


Figure 3.2: A modulo-8-counter, composed of three one bit counters.

such transitions in states `0` and `1` for condition $\overline{\texttt{a}}$.

Figure 3.2 shows a modulo-8-counter, which has been constructed with the *parallel product* and the *encapsulation*. The three automata whose states are respectively {`A0, A1`}, {`B0, B1`}, and {`C0, C1`} are put *in parallel*: they are separated by dashed lines. The rectangular box with the cartridge at its bottom is the graphical syntax for the encapsulation. The signals in the cartridge, `b` and `c`, are declared local to the program in the box. They can be neither written nor read from the outside of the box. Each signal is used as input by some of the parallel automata, and it is used as output by others: a communication will take place between the emitting and the reading automata.

Figure 3.2 constructs a modulo-8-counter by putting three 1-bit counters in parallel, and using encapsulation for the carry signals `b` and `c`. The lowest automaton emits a `b` every two `a`'s, the one in the middle emits a `c` every two `b`'s, and the upper automaton emits an `end` every two `c`'s. The complete program thus emits an `end` every eight `a`'s.

We now extend the modulo-8-counter with two additional features, a "stop counting" button, which interrupts the counting process, and the possibility to turn the counter completely off.

The "stop counting" button `sc` interrupts the counting process while it is pressed, and the counting resumes where it stopped when the button is released. This is implemented with *inhibition*, which is another unary operator: the notation is again a cartridge below a rectangular box, with the cartridge containing a fresh variable between "<" and ">". *Fresh* means that the inhibition signal must not be used in the inhibited program.

To have the possibility to turn the counter completely off, we give an automaton with two states `Counting` and `Not Counting` in Figure 3.3. The `Counting` state is said to be *refined*, because it contains the modulo-8-counter. Each time the refined automaton leaves state `Counting`, the refining program (i.e. the modulo-8-counter) is killed, and is created from scratch when the automaton comes back to `Counting` again, and thus starts counting from 0. Note that emitting `end` also finishes the counter, it goes to state `Not Counting`. When a program terminates itself by emitting a signal that makes the refined program leave the state, it is said to commit suicide.

In the graphical syntax, the *interface* of a program is implicitely defined. Inhibition variables and all signals which appear in a left-hand (resp. right-hand) side of a label, and are not declared to be local to some part of the program are *global inputs* (resp. *global outputs*). Together, they constitute the interface of the program. E.g., the program in Figure 3.1 has input `a` and output `b`, the program in Figure 3.2 has input `a` and output `end`, whereas the program in Figure 3.3 has inputs `a`, `sc`, `start`, and `stop`, and no outputs.

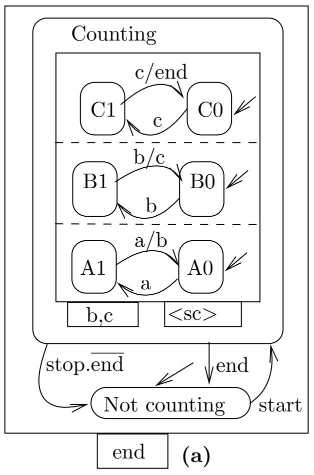


Figure 3.3: The modulo-8-counter, with an additional "stop counting" input `sc`, and `start` and `stop` buttons.

3.2 Formal Semantics

In this section, we define the semantics of Argos formally. We first define a trace semantics, which is common to all synchronous languages, then formally define flat automata, and finally define the Argos operators.

3.2.1 Traces and Trace Semantics

All synchronous languages have a common semantics based on traces. We here define it formally. A trace corresponds to one infinite execution of a program, and records the values of the inputs and the outputs at each instant, but does not contain the internal state of the program. It can therefore be defined independently of any programming language.

Definition 1 (Traces)**.** *Let $\mathcal{I}$, $\mathcal{O}$ be sets of Boolean input and output variables representing signals from and to the environment. An* input trace*, it, is a function: $it : \mathbb{N} \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. An* output trace*, ot, is a function: $ot : \mathbb{N} \longrightarrow [\mathcal{O} \longrightarrow \{\texttt{true}, \texttt{false}\}]$. We denote by InputTraces (resp. OutputTraces) the set of all input (resp. output) traces over $\mathcal{I}$ (resp. $\mathcal{O}$). 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 $n \in \mathbb{N}$.*

By abuse of notation, we sometimes write $it(n)$ for $\{i | it(n)[i] = \texttt{true}\}$ and $ot(n)$ for $\{o | ot(n)[o] = \texttt{true}\}$. A set of traces is a way to define the semantics of a program P, given its inputs and outputs. We can now define equality of single traces, and then determinism and completeness of a set of i/o-traces.

Definition 2 (Equality of Traces)**.** *Two traces t_1 and t_2 are equal, noted $t_1 = t_2$, if they range over the same set of variables V, and $\forall n \in \mathbb{N} \,.\, \forall v \in V \,.\, t_1(n)[v] = t_2(n)[v]$. Two pairs of i/o traces (it_1, ot_1) and (it_2, ot_2) are equal if $it_1 = it_2$ and $ot_1 = ot_2$.*

Definition 3 (Determinism and Completeness). *A set of pairs of i/o-traces* $S = \{(it, ot) \mid it \in \mathit{InputTraces} \wedge ot \in \mathit{OutputTraces}\}$ *is* deterministic *iff* $\forall (it, ot), (it', ot') \in S \,.\, (it = it') \implies (ot = ot')$*, and it is* complete *iff* $\forall it \in \mathit{InputTraces} \,.\, \exists ot \in \mathit{OutputTraces} \,.\, (it, ot) \in S$*.*

From the above definition, a program P is *deterministic* if from the same sequence of inputs it always computes the same sequence of outputs. It is *complete* whenever it allows every sequence of every valuations of inputs to be computed.

Finally, we define a trace combination operator, which combines an input and an output trace to a single trace. It will be useful later in the document.

Definition 4 (Trace Combination). *Let* $it : \mathbb{N} \longrightarrow [\mathcal{I} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *and* $ot : \mathbb{N} \longrightarrow [\mathcal{O} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *be traces, with* $\mathcal{I} \cap \mathcal{O} = \emptyset$*. Then,* $it.ot : \mathbb{N} \longrightarrow [\mathcal{I} \cup \mathcal{O} \longrightarrow \{\mathtt{true}, \mathtt{false}\}]$ *is a trace such that* $\forall i \in \mathcal{I} \,.\, it.ot(n)(i) = it(n)(i) \wedge \forall o \in \mathcal{O} \,.\, it.ot(n)(o) = ot(n)(o)$*.*

3.2.2 Automata

We now define flat automata, their semantics, and determinism, completeness, and equivalence for them.

We first define Boolean formulas. Instead of the dot which we use for conciseness in the automata to denote conjunction, we use a more conventional $\wedge$.

Definition 5 (Boolean Expressions). *A Boolean expression* B *over variables* $\mathcal{V}$ *is defined by the following grammar:*

$$\begin{array}{rll} B ::= & B \wedge B & \textit{conjunction} \\ \mid & B \vee B & \textit{disjunction} \\ \mid & \overline{B} & \textit{negation} \\ \mid & v & v \in \mathcal{V} \,. \end{array}$$

$\mathcal{B}ool(\mathcal{V})$ *denotes the set of Boolean expressions with variables in* $\mathcal{V}$*.*

We can now formally define an automaton.

Definition 6 (Automaton). *An* automaton A *is a tuple* $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *where* $\mathcal{Q}$ *is the set of states,* $s_0 \in \mathcal{Q}$ *is the initial state,* $\mathcal{I}$ *and* $\mathcal{O}$ *are the sets of Boolean input and output variables respectively,* $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times 2^{\mathcal{O}} \times \mathcal{Q}$ *is the set of transitions. For* $t = (s, \ell, O, s') \in \mathcal{T}$*,* $s, s' \in \mathcal{Q}$ *are the source and target states,* $\ell \in \mathcal{B}ool(\mathcal{I})$ *is the triggering condition of the transition, and* $O \subseteq \mathcal{O}$ *is the set of outputs emitted whenever the transition is triggered. Without loss of generality, we consider that automata only have complete monomials as input part of the transition labels.*

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

The semantics of an automaton is defined by the set of traces produced by all its executions.

Definition 7 (Semantics of an Automaton)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton. Its semantics is given in terms of a set of pairs of i/o-traces, Traces(A). This set is built using the following functions:*

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

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

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

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

$$(it, ot) \in \textit{Traces}(A) \iff \\ \forall n > 0\ .\ \forall o \in \mathcal{O}\ .\ o \in O\_step(s_0, it, n) \iff ot(n-1)(o)\ . \quad (3.1)$$

Equation (3.1) states that at each step n, the outputs emitted when executing the input trace *it* from the initial state are exactly the ones of *ot*.

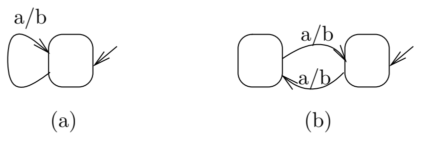
The automaton A is said to be *deterministic* (respectively *complete*) iff its set of traces *Traces*(A) is deterministic (resp. complete), as defined in Definition 3. We can give an equivalent definition using the determinism and the completeness of states as defined by Definition 8.

Definition 8 (State Determinism and Completeness)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ and $s \in \mathcal{Q}$ a state. s is* complete *iff $\bigwedge_{(s,\ell,O,s')\in\mathcal{T}} \ell = \texttt{true}$ and* deterministic *iff $(s, \ell_1, O_1, s'_1) \in \mathcal{T} \wedge (s, \ell_2, O_2, s'_2) \in \mathcal{T} \Rightarrow (\ell_1 \wedge \ell_2 = \texttt{false} \vee (O_1 = O_2 \wedge s'_1 = s'_2)$.*

An automaton is complete and deterministic if all its reachable states are so. We can now define a semantic equivalence between automata.

Definition 9 (Trace Equivalence)**.** *Two automata A_1, A_2 are* trace-equivalent*, noted $A_1 \sim A_2$, iff Traces(A_1) = Traces(A_2).*

As an example for equivalent automata, consider the automata in Figure 3.4(a) and (b). Both have the same set of traces, namely all traces where `a` and `b` occur at the same time, and are thus equivalent.



(a) (b)

Figure 3.4: Two equivalent automata.

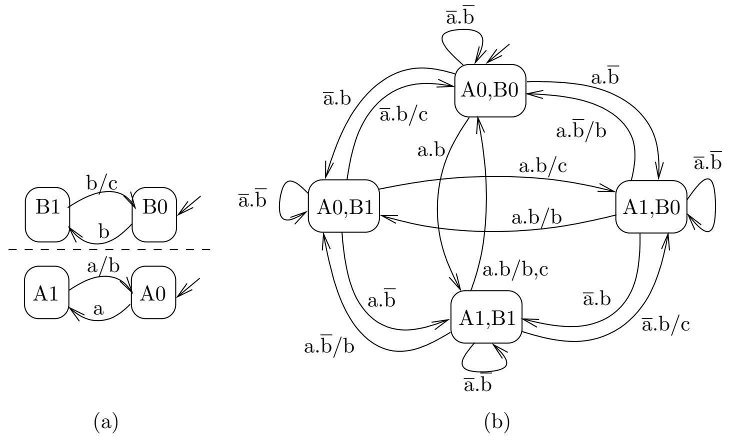


Figure 3.5: The parallel product of two one bit counters. The hierarchical automaton is shown to the left, and the flat automaton to the right.

3.2.3 Argos Operators

We now formally define the four existing operators of Argos by giving translations to flat automata, starting with the synchronous product.

Definition 10 (Synchronous Product). *Let* $A_1 = (\mathcal{Q}_1, s_{01}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ *and* $A_2 = (\mathcal{Q}_2, s_{02}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ *be automata. The* synchronous product *of* A_1 *and* A_2 *is the automaton* $A_1 \| A_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, (s_{01}, s_{02}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T})$ *where* $\mathcal{T}$ *is defined by:*

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

The synchronous product of automata is both commutative and associative, and it is easy to show that it preserves both determinism and completeness. As an example, consider Figure 3.5 (b). It shows the parallel product of the two lower one bit counters from Figure 3.2, shown again in Figure 3.5 (a).

Next, we define encapsulation, which introduces local signals, that can be used for communication between parallel automata.

Definition 11 (Encapsulation). *Let* $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ *be a set of inputs and outputs of* A*. The* encapsulation *of* A *with respect to* Γ *is the automaton* $A \setminus \Gamma = (\mathcal{Q}, s_0, \mathcal{I} \setminus \Gamma, \mathcal{O} \setminus \Gamma, \mathcal{T}')$ *where* $\mathcal{T}'$ *is defined by:*

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

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

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

If the label of a transition satisfies this criterion, then the names of the encapsulated variables are hidden, both in the input part and in the output part. This is expressed by

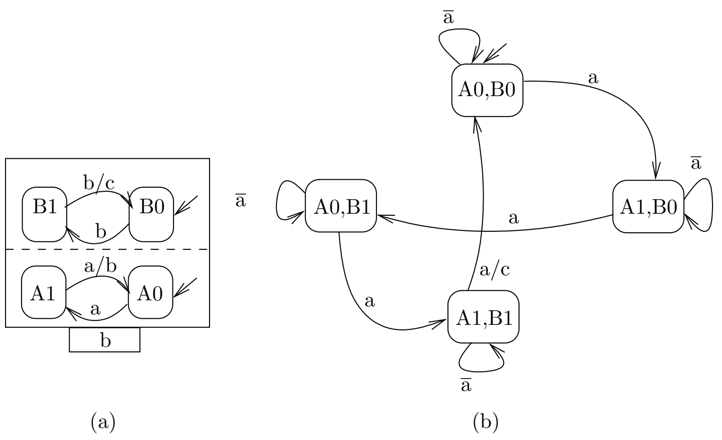


Figure 3.6: An hierarchical 2-bit counter to the left, that is flattened into the automaton to the right.

$\exists\Gamma\ .\ \ell$ for the input part, and by $O \setminus \Gamma$ for the output part. $\exists\Gamma\ .\ \ell$ is formally defined as $\exists\Gamma\ .\ \ell = \bigwedge_{a\in\ell^+\setminus\Gamma} a \wedge \bigwedge_{a\in\ell^-\setminus\Gamma} \overline{a}$.

Figure 3.6 (b) shows the encapsulation of Figure 3.5 by `b`. Note that only the transitions whose conditions fulfill the criterion for `b` are still there, and that `b` has disappeared from their conditions. Figure 3.6 (a) shows the hierarchical program that is flattened in Figure 3.6 (b).

In general, the encapsulation operation does not preserve determinism nor completeness. As an example, consider the product of two automata in Figure 3.7 (a). Both automata are deterministic and complete, their product, encapsulated by `a,b` in Figure 3.7 (b), however, is neither. The combination of state `A` with state `0` is not deterministic. When `i` is true, two reactions are possible: either transition `i.ā` in state `A` and transition `i.b̄` in state `0` or transition `i.a/b` in state `A` and transition `i.b/a` in state `0`. The combination of state `A` with state `1`, on the other hand, is not complete, no reaction is possible when `i` is true.

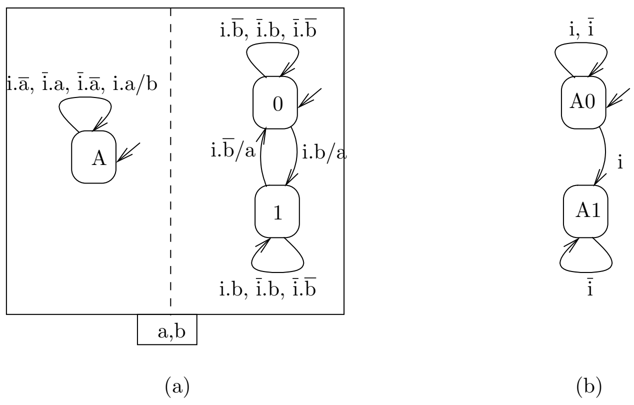


Figure 3.7: An example of the encapsulation preserving neither determinism nor completeness.

This is related to the so-called "causality" problem intrinsic to synchronous languages (see, for instance [BG92]). It occurs when parallel automata depend on each other's outputs

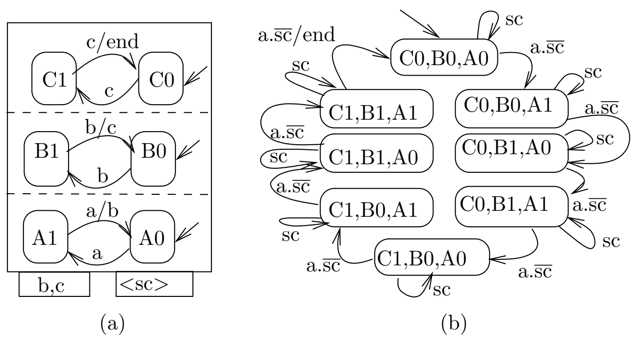


Figure 3.8: The 3-bit counter inhibited by `sc` is shown in the left automaton, the result of its flattening in the right one.

in the same instant, forming a circular dependency, and there is not exactly one possible valuation for the encapsulated signals. In practice, however, such cases are rare: communicating components do not usually form circular dependencies between each other. Lustre forbids them altogether, and this is not a major constraint for programmers.

The inhibition operator is used to momentarily freeze the Argos component to which it is applied.

Definition 12 (Inhibition)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $a \notin \mathcal{I}$ a signal. The inhibition of A by a is defined as A `whennot` $a = (\mathcal{Q}, s_0, \mathcal{I} \cup \{a\}, \mathcal{O}, \mathcal{T}')$, where $\mathcal{T}'$ is defined by*

$$(s, \ell, O, s' \in \mathcal{T}) \Rightarrow (s, \ell \wedge \overline{a}, O, s') \in \mathcal{T}' \wedge (s, \ell \wedge a, \emptyset, s) \in \mathcal{T}' \ .$$

Figure 3.8 shows the automaton of Figure 3.2 with the inhibition by `sc` applied to it. Each transition has $\overline{\texttt{sc}}$ added to its condition, and each state has a new loop transition with `sc` as condition and no outputs.

The refinement operator refines the behavior of states by putting Argos automata into them.

Definition 13 (Refinement)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton with states $\mathcal{Q} = \{s_0, \dots, s_n\}$, and let $A_i = (\mathcal{Q}_i, s_{0i}, \mathcal{I}_i, \mathcal{O}_i, \mathcal{T}_i), i = 0 \dots n$ be automata with states $\mathcal{Q}_i = \{s_{0i} \dots s_{n_i i}\}$, which refine the states of A. The refinement of A by $\{A_i\}_{i=0\dots n}$ is defined by $A \triangleright \{A_i\}_{i=0\dots n} = (S \triangleright \{A_i\}_{i=0\dots n}, s_0 \triangleright s_{00}, \mathcal{I} \cup \bigcup \mathcal{I}_i, \mathcal{O} \cup \bigcup \mathcal{O}_i, T')$, where the refined set of states is defined by*

$$S \triangleright \{A_i\}_{i=0\dots n} = \{s_i \triangleright s_{ji} | i \leq n, j \leq n_i\}$$

and $\mathcal{T}'$ is defined by the following equations

$$(s_i, \ell, O, s_{i'}) \in \mathcal{T} \wedge (s_{ji}, \ell', O', s_{j'i}) \in \mathcal{T}_i \Rightarrow (s_i \triangleright s_{ji}, \ell \wedge \ell', O \cup O', s_{i'} \triangleright s_{i'0}) \in \mathcal{T}' \tag{3.2}$$

$$(s_{ji}, \ell', O', s_{j'i}) \in \mathcal{T}_i \Rightarrow (s_i \triangleright s_{ji}, \ell' \wedge \bigwedge_{(s_i, \ell, O, s_{i'})} \overline{\ell}, O', s_i \triangleright s_{j'i}) \in \mathcal{T}' \ . \tag{3.3}$$

When a transition $(s_i, \ell, O, s_{i'})$ in A is taken, the instance of A_i, i.e. the automaton that refines s_i, is killed at the end of the instant, and an instance of $A_{i'}$ starts executing in the next instant. However, the outputs emitted by A_i are still emitted. This is expressed by

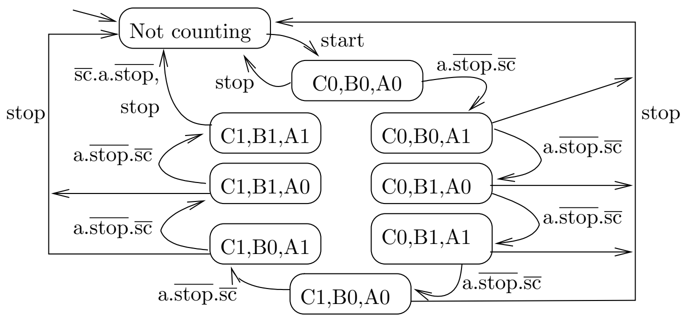

Not counting
start
$\overline{\text{sc}}$.a.$\overline{\text{stop}}$,
stop
stop
C0,B0,A0
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
stop
stop
C1,B1,A1
C0,B0,A1
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
C1,B1,A0
C0,B1,A0
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
C1,B0,A1
C0,B1,A1
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
a.$\overline{\text{stop}}$.$\overline{\text{sc}}$
C1,B0,A0

Figure 3.9: The flattened automaton of Figure 3.3.

Equation (3.2): the transition in T' goes to the initial state of $A_{i'}$, but emits both the outputs of $(s_i, \ell, O, s_{i'})$ and the outputs of the transition of A_i, O'.

If there is no transition in A to take, only the transition the refining automaton is taken. This is expressed by Equation (3.3).

The inhibition and refinement operators both preserve determinism and completeness. Figure 3.9 shows the automaton from Figure 3.3, with the definition of the operators applied to it.

Finally, we define a grammar for Argos.

Definition 14 (Grammar of Argos)**.** *The set of Argos expressions is defined by the grammar:*

$$\begin{array}{rll} P ::= & P \| P & \textit{parallel product} \\ | & P \setminus \Gamma & \textit{encapsulation} \\ | & P \ \texttt{whennot} \ a & \textit{inhibition} \\ | & A \triangleright \{R_i\}_{i=0\ldots n} & \textit{refinement} \\ R ::= & P \quad | \quad \texttt{NIL} & \textit{refining objects,} \end{array}$$

where A is an automaton as defined in Definition 6, Γ a set of signals, and a a signal. `NIL` *represents a state that is not refined.*

3.3 Synchronous Observers

A very useful feature of synchronous languages, which we will use in this thesis, is that they allow to define *observers* in a natural and simple way [HLR93]. Intuitively, an observer is a program that observes the behavior of another program by reading its inputs and outputs, and without modifying its behavior. It can compute any safety property (in the sense of safety/liveness properties as defined in [Lam77]).

An Argos observer is thus a program which has as inputs the inputs and outputs of the observed program, and usually a single output which indicates if the observed safety property is true. The observer is composed with the program in parallel, and the observed outputs are encapsulated. It can then be checked if the safety property is true for a certain sequence of inputs, or, by looking at the structure of the program, if it is always true. Thus, observers are well suited for testing and verification of synchronous programs.

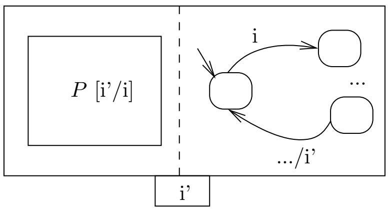


Figure 3.10: Simulating before advice in Argos.

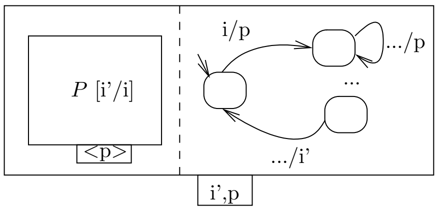


Figure 3.11: Simulating before advice in Argos, with inhibition.

3.4 Some Examples of Encoding Aspects

In Section 2.3.1, we claimed that some of the functionalities that need an aspect in sequential languages can be encoded with the parallel product in synchronous ones. In this section, we show some examples to support this claim, and also discuss limitations of the approach.

As a first example, consider a program P with inputs $\mathcal{I}$ and outputs $\mathcal{O}$. We now want to modify P in such a way that when P receives some input $i \in \mathcal{I}$, it emits an output o. Such a kind of modification could e.g. be used to log the occurrence of certain events, and is often implemented with an aspect in sequential languages. In Argos, this can be done easily by putting an automaton in parallel, which emits o when it receives i.

The parallel product alone allows us only to implement aspects that do not influence the execution of the base program, called spectative aspects [Kat06]. We can, however, implement more powerful aspects using other operators. E.g., we can implement a kind of `before` advice, by retaining inputs while the advice is executed. To insert advice before an input i, we let P accept a new input i' instead of i. Then, whenever i is true, we first emit our advice before emitting i', which is encapsulated. Figure 3.10 illustrates this example. After receiving i, we execute some Argos code (denoted by the dots on the right), before emitting i' and returning to the initial state.

Note that in the modification presented above, P continues executing during the execution of the advice, we only prevent it from receiving i as input. P thus continues updating its state and emitting outputs based on its other inputs, which are not delayed. This may not be desirable in certain situations. In such cases, we can interrupt the execution of P by inhibiting it while the advice executes. This is illustrated in Figure 3.11. Note that while the advice is executing, it must emit the inhibition signal p all the time, otherwise P will continue executing.

With inhibition, we can also implement a kind of `around` advice, with `proceed`. If the advice wants the program to proceed, it stops emitting p. It can then also emit i', thus relaying the intercepted signal i to P. Of course, P never terminates by itself, as does a call to `proceed` in sequential aspect languages, but the advice can observe the in- and outputs,

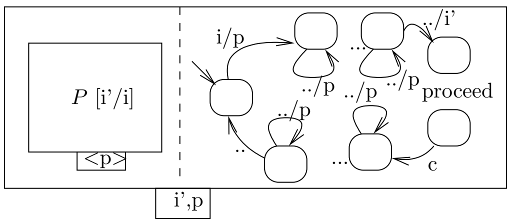


Figure 3.12: Simulating around advice in Argos.

and can inhibit P and continue emitting advice when it chooses. Figure 3.12 illustrates this concept of `around` advice. When i becomes true, the program is inhibited, and the advice starts executing, denoted by the two states in the top of Figure 3.12. Then, the advice can stop emitting p, and also emit i', the input that replaced i, thus letting P execute the `proceed`. When some condition c becomes true, the advice ends the `proceed` by emitting p, and starts executing advice again. When the aspects has finished execution, it returns to the initial state and stops emitting p.

In the above examples, we have always used an input as a join point. We can also use an output o in the same way, although we have to use an encapsulated copy o', which can be read by the parallel program. o' can also be used to delay emitting o. We can even describe an arbitrary safety property over the in- and outputs with an observer (see Section 3.3).

Using an output o to define join points, however, has a limitation when combined with inhibition. We cannot immediately inhibit the execution of P depending on o. That would also prevent o from being emitted, and thus lead to an incomplete program, where no reaction for o is defined. Thus, when we want to activate advice for o, we must let P finish the step that produced o, and start executing the advice only in the following step.

Thus, the examples show that we can implement some typical aspect-oriented tasks with the Argos operators, without modifying the base program. Indeed, the modifications presented in this section resemble to some extent the aspect-oriented Composition Filters approach [BA04], which intercepts and modifies messages between components. In the next chapter, we will introduce some modifications that cannot be expressed with the existing operators alone, and that thus need an aspect language.

3.5 Conclusion on Argos

Argos is a language with Mealy automata as base elements, which can be combined with a set of operators. It is a useful minimal synchronous languages: with four operators, it is reasonably small, but expressive enough to model meaningful programs, as we will see in the following chapters.

The semantics of the operators is purely synchronous, and we defined it formally, as the translation of an automaton built with an operator into a flat automaton. The semantics of flat automata is in turn defined by the set of traces they produce. The operators also have some nice properties: except for encapsulation, they preserve determinism and completeness of programs, and all operators preserve semantic equivalence between programs (see [Mar92] for a proof). Argos programs are thus strongly encapsulated, only their interface is visible from the outside, but not their internal structure. Thus, we can always replace any part of an Argos program with a semantically equivalent one, without changing the semantics of the program. The parallel composition is very expressive: many typical examples of aspects in sequential languages can be expressed with it in Argos. There are, however, remaining cross-cutting concerns, as we will see in the next chapter.

Chapter 4

Larissa

4.1 Designing an Aspect Extension for Argos

This chapter introduces Larissa, an aspect-oriented extension for Argos. An earlier version of this work has been published in [AMS06a].

Requirements for Larissa. As an aspect extension for a synchronous language, Larissa should fulfill the requirements discussed in Section 2.3.1. Thus, Larissa must be able to express recurrent cross-cutting concerns that occur in synchronous programs written in Argos. It must add additional expressiveness to Argos, and also model concerns that cannot be expressed with the parallel product. It must also fulfill the requirements imposed by the formal nature of synchronous languages: it must preserve determinism and completeness, and most importantly the equivalence between programs. This last point entails important restrictions: aspects must not refer to the to implementation details of the base program, but only its semantics.

Furthermore, Larissa should integrate well into Argos. This can be achieved best if aspect weaving can be considered another operator of Argos, along with e.g. the parallel product and the encapsulation. Besides preserving the equivalence, programmers should therefore be able to use aspects anywhere in the structure of their program, just as the other operators, and the use of aspects should also be invisible from the outside. Finally, Larissa aspects should be expressed in a way familiar to Argos programmers.

Elements of Larissa. As the other operators in Argos, we apply Larissa aspects to flat automata. Like aspects in other languages, Larissa aspects consist of a pointcut and a piece of advice. Join points denote well-defined points in the execution of the program. Such points correspond to transitions in Argos programs, which represent the step a program makes during one instant. Transitions seem thus well suited as join points. Note that a transition groups many different steps, because it can be taken several times during the same execution, under different circumstances. However, pointcuts may wish to select only a subset of the steps a transition represents. In such a case, the weaving process splits the transition into different transitions, to separate the steps chosen by the pointcut from the other steps. We call the transitions identified by the pointcut *join point transitions.*

Indeed, specifying join points independently of the transitions of the base program is necessary to respect the encapsulation of the base program. Which steps of the execution are grouped in one transition is an implementation choice, and thus aspects must not depend on it.

Once a pointcut has identified the join point transitions, a piece of advice changes them into *advice transitions.* To respect the requirements given above, advice must not modify the join point transitions arbitrarily. A transition consists of a source and a target state, a condition, and the emitted outputs. The advice does not modify the source state and the condition of the join point transitions, because these are part of the steps chosen as join points by the pointcut, and must thus not be changed. Furthermore, leaving the source state and the condition preserves determinism and completeness of the base program.

However, the advice changes the outputs and target state of the transitions. These elements change the behavior of the program in the current and future steps. To preserve the equivalence, this modification is also specified in terms of the program semantics, and not in terms of the implementation.

In this chapter, we will introduce Larissa as an aspect language that follows these lines, using a running example.

Introductory Examples. This chapter introduces two different kinds of aspects. Each aspect is first presented with an example, and then formally defined as a translation into a flat automaton, in the same way as the other Argos operators.

The examples for both kinds of aspects are extracted from the same case study, a juice processing plant [FvS99]. The plant is divided into three departments. The first one produces the juice, the second one pasteurizes it and the last one packages it. As we discussed above, Larissa only depends on the interface of its base program, and not on its implementation details. Therefore, it should be possible to write Larissa aspects without knowing the implementation of the base program, but by referring only to the interface, and an informal description of its semantics. We will adopt this approach in the introductory examples in this chapter.

The introductory examples are structured as follows. We first describe the physical system to be controlled, and then the interface of its controller, i.e. its inputs and outputs. The inputs of the controller are signals from sensors in the plant (e.g. “tank full”) and commands from the user (e.g. “start pasteurizing”), and the outputs of the controller are commands that are sent to the plant (e.g. “close a valve”). This interface is precisely known and well documented, but the implementation of the program is not known. 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. Finally, we give a sample controller and show how the additional functionality is added by weaving the aspect.

After introducing each aspect informally in the example, we define it formally. Formal properties of the aspects are described and proved in Chapter 6.

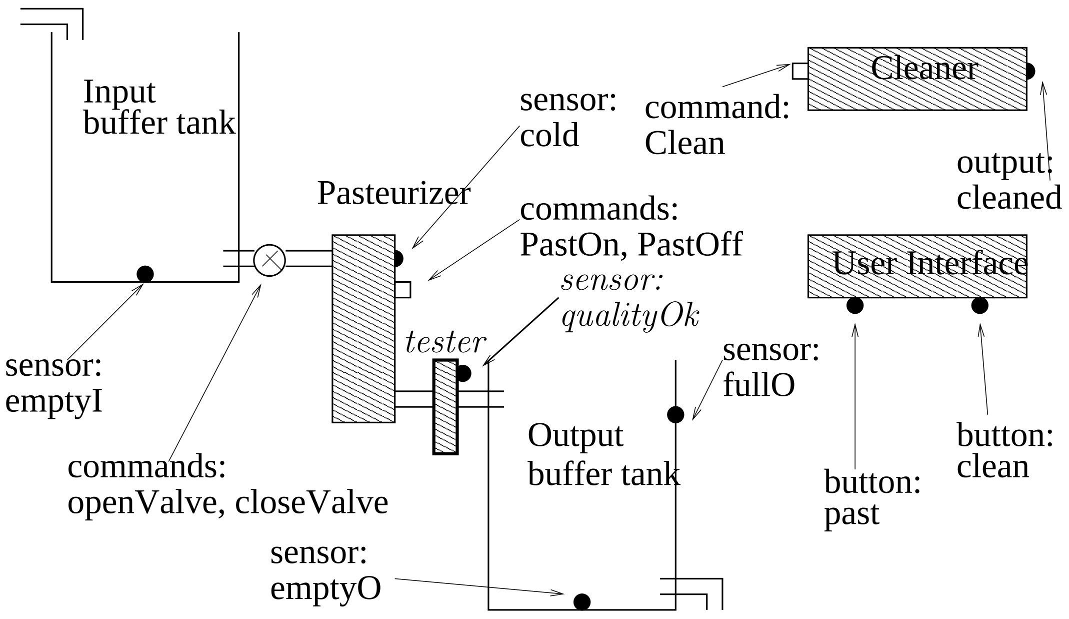


Figure 4.1: A simplified view of the pasteurization department

4.2 A first Kind of Aspects: *toInit* and *toCurrent*

Section 4.2.1 introduces aspects for Argos using an example taken from the pasteurization department of a juice factory. Section 4.2.2 discusses some extensions, and Section 4.2.3 defines aspects formally.

4.2.1 The Pasteurizer Controller

4.2.1.1 The Pasteurizer

A simplified view of the pasteurization department of the juice factory is shown in Figure 4.1. The unpasteurized juice comes from the input buffer tank. When the valve is open, the juice continuously flows to the pasteurizer where it is heated and cooled down several times. After leaving the pasteurizer, it flows to the output buffer tank. Occasionally, the pasteurizer and the buffer tanks must be cleaned by a cleaning unit which is part of the pasteurization department.

4.2.1.2 The Controller Specification

The user interface panel contains two buttons, one for pasteurizing, the other for cleaning the pasteurizer. 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 cleaning unit is working. The machine has various sensors and actuators. The sensors transmit Boolean signals to 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 Figure 4.2.

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

Figure 4.2: Meaning of the sensors, buttons and commands.

4.2.1.3 Adding a Quality Tester to the Controller

We now want to add a quality tester to the department. It will be placed between the pasteurizer and the output buffer tank to test the quality of the juice. It is shown in italic in Figure 4.1. It is a sophisticated sensor whose signal `qualityOk` is false when a quality problem is detected and true otherwise.

When the tester detects a problem, we should stop the pasteurization process, and clean the pasteurizer. This new behavior has to be added to the normal behavior of the controller. We can specify it precisely as follows: *"If `qualityOk` is false while pasteurizing, then stop the pasteurization process and start cleaning"*.

The new behavior could be implemented in the following way: 1) identify points in the execution of the program where the condition *"If `qualityOk` is false while pasteurizing"* holds, 2) place instructions at these points that *"stop the pasteurization process and start cleaning"*. In an Argos program, the program points identified by 1) are transitions, which must be replaced by other transitions which fulfill the specification of 2).

In the sequel, we propose to use the specification *"If `qualityOk` is false while pasteurizing, 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, but not its internal structure.

4.2.1.4 A Notion of Aspect

To perform the kind of transformation described above for the controller, we must change some transitions. Therefore, we need to specify: (1) which transitions we modify, this corresponds to the *pointcut* of the aspect. (2) how to modify the selected transitions, this corresponds to the *advice* of the aspect. For this example, we first specify the advice, and then the pointcut.

Specifying the advice. In this example, the advice specifies the new outputs and the new target state of the selected transitions. This is sufficient, because only individual transitions have to be changed, and the source state and the triggering condition are selected by the pointcut, so changing these could make the program non-deterministic and incomplete. In the example, the controller has to emit the outputs `Clean` (to start cleaning), `PastOff` and `CloseValve` (to stop the pasteurization process) when the aspect is activated. To define the target state of the advice transitions, we use a finite sequence of input values, which we call the input trace σ. It uniquely determines a state in a complete and deterministic program: it is the state that would be reached by executing σ from the initial state of the program.

We call this kind of advice a *toInit advice*: all advice 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. 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 works can be reached from the initial state in one step if the input `clean` is true, i.e. by the finite input trace $\sigma = (clean)$.

Selecting join points with a *pointcut*. The join points are the transitions where the behavior must be modified, called join point transitions. We have to specify transitions in a system whose internal structure is not known. We cannot repeat the trick we used to select the target state (a finite sequence of inputs, to be played from the initial state) here, because it would break the preservation of the behavior equivalence. 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 $\sigma' = (a.a)$: modifying the transition chosen by σ' in A_1 and in A_2 leads to non equivalent automata. (Note 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 point transitions with a special Argos program, called the *pointcut*. It is a synchronous observer (see Section 3.3) of the base program, that emits a single fresh output `JP` each time a join point is reached. An observer as pointcut has the advantage of describing the join points independently of the program. The synchronous product between the base program and pointcut separates the transitions selected by the pointcut from those that are not selected, and marks them with `JP`. This way of selecting the join point transitions respects the encapsulation: it uses only the parallel product and encapsulation, which both respect it.

In the example, the transitions we need to replace are the transitions where the pasteurizer is working and `qualityOK` is false. The pointcut listens to the inputs and outputs of the program in order to deduce when such a transition is taken. To decide whether the pasteurizer is working, the pointcut listens to the commands issued to the actuators, in this case the commands `PastOn` and `PastOff`. The automaton in Figure 4.3 emits `JP` when the pasteurizer is working and `qualityOK` is false, and thus selects all the corresponding transitions in the base program.

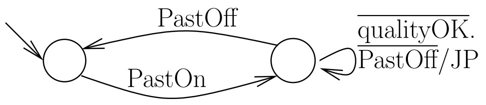


Figure 4.3: Pointcut program for the quality tester.

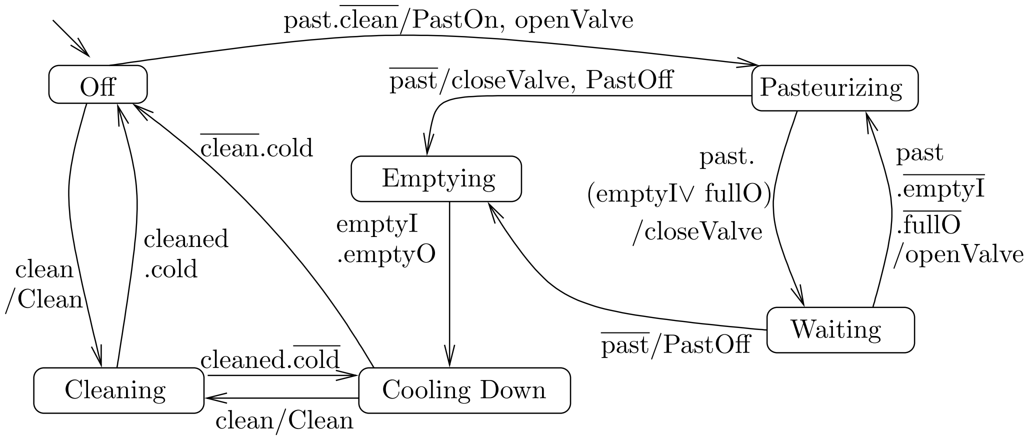

past.$\overline{\text{clean}}$/PastOn, openValve
Off
$\overline{\text{past}}$/closeValve, PastOff
Pasteurizing
$\overline{\text{clean}}$.cold
Emptying
past.
(emptyI∨ fullO)
/closeValve
past
.$\overline{\text{emptyI}}$
.$\overline{\text{fullO}}$
/openValve
emptyI
.emptyO
cleaned
.cold
clean
/Clean
Waiting
$\overline{\text{past}}$/PastOff
cleaned.$\overline{\text{cold}}$
Cleaning
Cooling Down
clean/Clean

Figure 4.4: A sample pasteurizer controller.

4.2.1.5 A sample controller

Figure 4.4 shows a sample controller. It can either be cleaning or pasteurizing. The pasteurizing is suspended for a while by closing the valve if the input buffer tank is empty or the output buffer tank is full (state `Waiting`). 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.

Note 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`. Thus, the parallel product with the pointcut does not create new states. All outgoing transitions of the states `Pasteurizing` and `Waiting` where `qualityOK` is false are selected by the pointcut in Figure 4.3. These transitions are not explicitly present in Figure 4.4, but they can be easily created by replacing each outgoing transition of these states by two transitions, one with an additional `qualityOK` in the condition, and the other with $\overline{\texttt{qualityOK}}$.

The trace 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 replace the selected join point transitions with advice transitions pointing to the `Cleaning` state. These transitions will have outputs `PastOff, CloseValve`, and `Clean`.

Weaving an aspect into the controller has modified its interface: the input `qualityOk` has been added. The woven controller is given in Figure 4.5. Modified transitions and conditions are shown in bold.

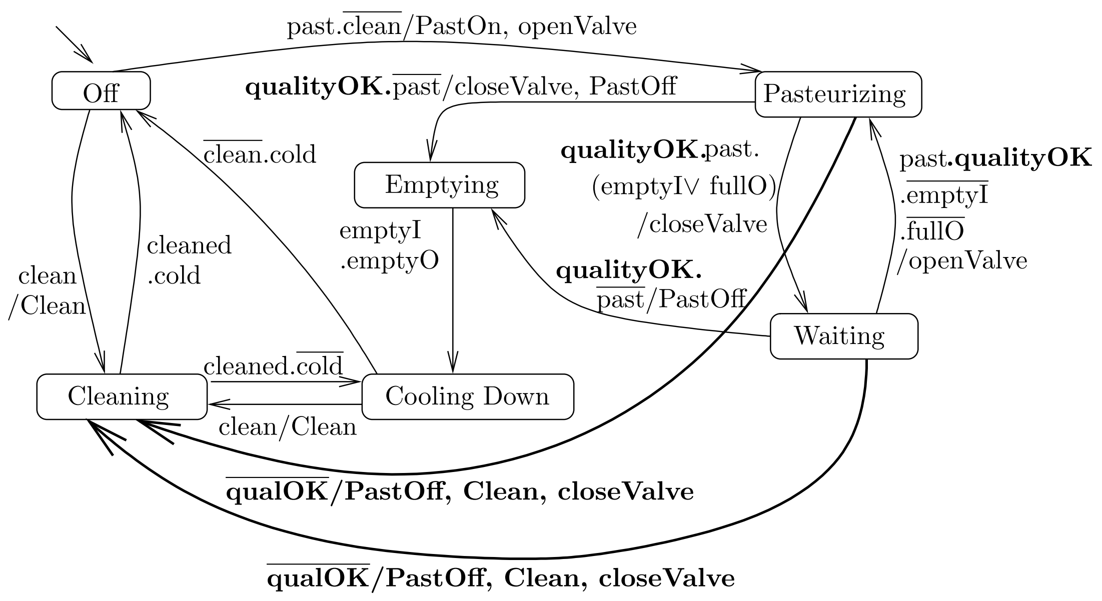

past.$\overline{\text{clean}}$/PastOn, openValve
Off
Pasteurizing
qualityOK.$\overline{\text{past}}$/closeValve, PastOff
$\overline{\text{clean}}$.cold
Emptying
qualityOK.past.
(emptyI∨ fullO)
/closeValve
past**.qualityOK**
.$\overline{\text{emptyI}}$
.$\overline{\text{fullO}}$
/openValve
emptyI
.emptyO
cleaned
.cold
clean
/Clean
qualityOK.
$\overline{\text{past}}$/PastOff
Waiting
cleaned.$\overline{\text{cold}}$
Cleaning
Cooling Down
clean/Clean
$\overline{\text{qualOK}}$/PastOff, Clean, closeValve
$\overline{\text{qualOK}}$/PastOff, Clean, closeValve

Figure 4.5: The sample pasteurizer controller from Figure 4.4, with the aspect woven into it.

4.2.2 Variants of the Aspect Language

This section discusses two variants of *toInit* advice that are not illustrated in the example in the previous section. An example of the first variant, *toCurrent* advice, can be found in Chapter 5. The second variant, advice programs, is introduced with a modification of the pasteurizer example from the previous section.

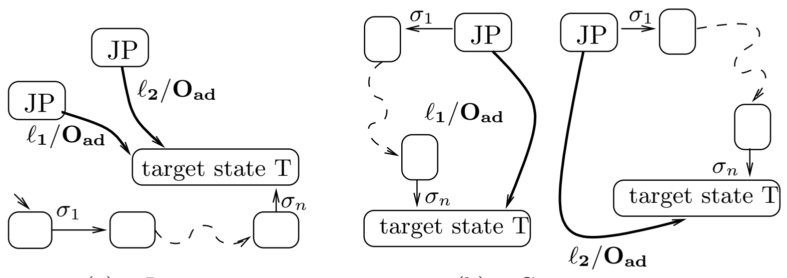
4.2.2.1 *toCurrent* Advice

With *toInit* advice, we always jump to a the same location for all advice transitions. However, we sometimes want to jump to different locations depending on the current join point transition. This is closer to the way other aspect languages work, where aspects usually return to the location in the code where they started (or remove a single method call), and where the state of the program is modified starting from state of selected join point.

Therefore, the first variant changes the way we select the target state using the trace σ. Instead of executing σ from the initial state of the automaton, it can be executed from the initial state of the join point transition. This kind of advice is called *toCurrent* advice. The mechanism is illustrated by Figure 4.6. Instead of jumping to a fixed location, this corresponds to jumping forward from the current location. As opposed to *toInit* advice, the location of the target state of the advice transition is not the same for all transitions, but depends on where the advice is activated. An example for *toCurrent* advice can be found in Chapter 5.

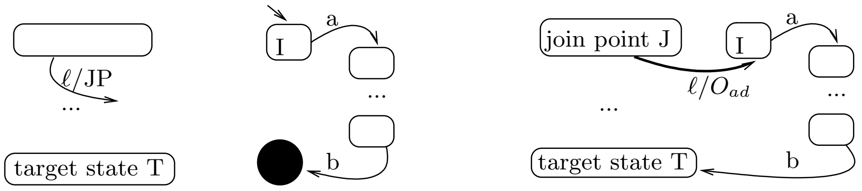
4.2.2.2 Advice Programs

Imagine that the pasteurizer must be empty when it is cleaned. Then, the quality tester from Section 4.2.1.3 must be modified, such that it empties the buffer before it goes to state `Cleaning`. This cannot be addressed by changing just the target state of the aspect, e.g. to



(a) toInit aspect (b) toCurrent aspect

Figure 4.6: Schematic *toInit* and *toCurrent* aspects. (Advice transitions are in bold.)



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

Figure 4.7: Inserting an advice program.

state `Emptying`, because we cannot force the controller to go to state `Cleaning` once it has emptied the tanks. Thus, we must modify the advice transitions, such that they do not go to `Cleaning` directly, but they must first execute some code which empties the buffer.

This is specified with an *advice program* P_{adv} that *terminates*. When an aspect is activated, the advice program is executed until it terminates, and then the woven program goes to the target state specified by the aspect. Since Argos has no built-in notion of termination, the programmer of the aspect has to identify a final state F in the advice program. We denote F by filled black circles in the figures.

Inserting an advice program is quite similar to inserting a transition. We first specify a target state T by a finite input trace, starting either from the initial state (*toInit*) or from the source state (*toCurrent*) of the join point transition. Note that there may be more than one target state for *toCurrent* advice. Then, for each T, a copy of the automaton P_{adv} is inserted, which means: 1) replace every join point transition J with target state T by a transition to the initial state I of the instance of P_{adv} associated with T. As for advice transitions, the input part of the label is unchanged and the output part is replaced by the *advice outputs* O_{adv}; 2) connect the transitions that went to the final state `F` in P_{adv} to T. This is illustrated by Figure 4.7.

In the example, we use such an advice program to empty the tanks, and then go to state `Cleaning`. The modified aspect then insert the advice program shown in Figure 4.8 and emits the advice outputs `PastOff` and `CloseValve`. Pointcut and trace are the same as in the aspect in the previous section. The woven program is shown in Figure 4.9.

4.2.3 Formal Definition

After we introduced the concepts of *toInit* and *toCurrent* advice informally, we define them formally in the same way as the Argos operators, by giving translations into flat automata.

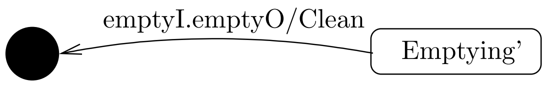


Figure 4.8: The advice program inserted to empty the pasteurizer before cleaning.

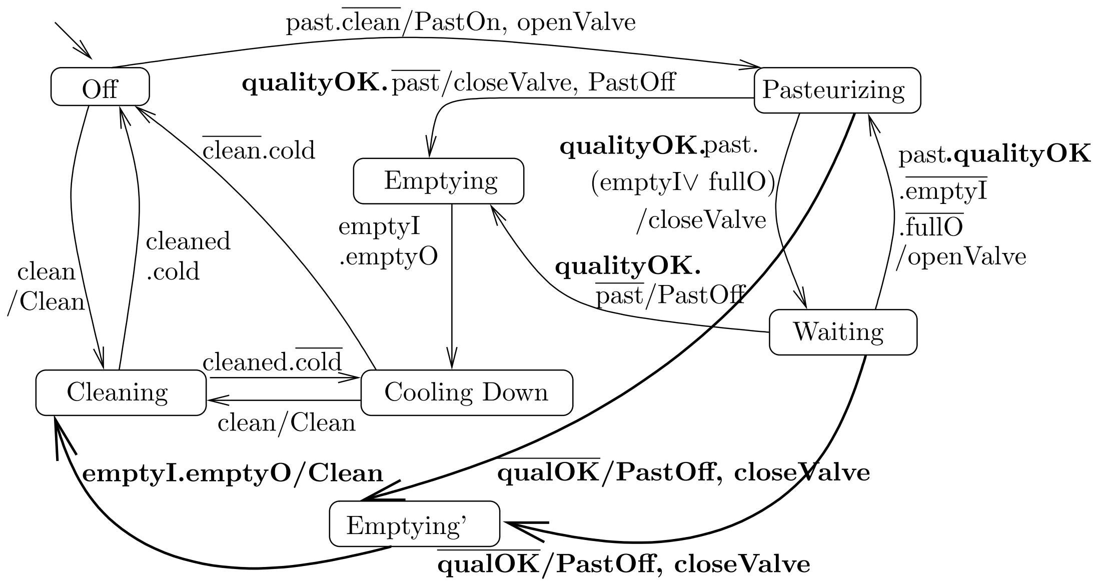

past.$\overline{\text{clean}}$/PastOn, openValve
Off
qualityOK.$\overline{\text{past}}$/closeValve, PastOff
Pasteurizing
$\overline{\text{clean}}$.cold
Emptying
qualityOK.past.
(emptyI∨ fullO)
/closeValve
past**.qualityOK**
.$\overline{\text{emptyI}}$
.$\overline{\text{fullO}}$
/openValve
emptyI
.emptyO
cleaned
.cold
clean
/Clean
qualityOK.
$\overline{\text{past}}$/PastOff
Waiting
cleaned.$\overline{\text{cold}}$
Cleaning
Cooling Down
clean/Clean
emptyI.emptyO/Clean
$\overline{\text{qualOK}}$/PastOff, closeValve
Emptying'
$\overline{\text{qualOK}}$/PastOff, closeValve

Figure 4.9: The sample pasteurizer controller from Figure 4.4, with an aspect woven into it that empties the buffer before starting the cleaning.

We first formally define an aspect, then we define how to select join point transitions, and finally how to weave *toInit* and *toCurrent* advice into a program, also taking into account advice programs.

Definition 15 (*toInit*/*toCurrent* Aspects)**.** *A toInit or toCurrent aspect, for a program P on inputs $\mathcal{I}$ and outputs $\mathcal{O}$, is a tuple (P_{JP}, adv) where*

- $P_{JP} = (\mathcal{Q}_{pc}, s_{0pc}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{pc})$ *is the pointcut program, and JP occurs nowhere else in the environment.*
- $adv = (type, O_{adv}, \sigma, P_{adv})$ *is the advice, which contains four items:*
  - $type \in \{toInit, toCurrent\}$ *defines how the target state of the transitions is determined, by executing the trace either from the initial or the current state.*
  - $O_{adv} \subseteq \mathcal{O}$ *is the set of outputs emitted by the advice transitions.*
  - $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *is a finite input trace of length $\ell_\sigma + 1$. It defines the single target state of the advice transitions by executing the trace either from the initial state or the current state.*
  - $P_{adv} = (\mathcal{Q}_{ins} \cup \{F\}, s_{0ins}, \mathcal{I}_{ins}, \mathcal{O}_{ins}, \mathcal{T}_{ins})$*, with $\mathcal{I}_{ins} \subseteq \mathcal{I}$ and $\mathcal{O}_{ins} \subseteq \mathcal{O}$, is the advice program, which is executed after every advice transition. The final state F designates the return point to the program. If this item is not given, we assume it to be $(\{F\}, F, \emptyset, \emptyset, \emptyset)$: we are then weaving advice transitions.*

In the definition, aspects do not have the right to add new in- and outputs to the program, but we did so in the examples. Although this is an important functionality of aspects, forbidding it in the definition is no restriction: we can always add new signals to the interface of the base program before we apply the aspect. Inputs added that way then have no effect on the program, and added outputs are never emitted. On the other hand, this restriction makes the formal definition of weaving easier.

4.2.3.1 Join Point Weaving

Join point transitions are selected in a program P by an observer P_{JP}, which has P's inputs and outputs as inputs, and has one output *JP*. A transition is selected as a join point transition if P_{JP} emits *JP* while the transition is taken during the parallel execution of P and P_{JP}.

If we simply put a P and P_{JP} in parallel, we must encapsulate P's outputs $\mathcal{O}$, so that P_{JP} can read them. They will become synchronization signals, and are thus no longer emitted by the product. We avoid this problem by introducing a new output o' for each output o of P: o' will be used for the synchronization with P_{JP}, and o will still be visible as an output. We duplicate each output of P by putting P in parallel with one single-state automaton per output o, which emits o if o' is true.

Definition 16 (Join Point Weaving)**.** *Let P be a program on inputs $\mathcal{I}$ and outputs $\mathcal{O} = \{o_1, ..., o_n\}$, and let P_{JP} a pointcut of an aspect for P. Then, let P' be P and P'_{JP} be P_{JP}, where $\forall o \in \mathcal{O}$, o is replaced by o'. Furthermore, let $dupl_o = (\{q\}, q, \{o'\}, \{o\}, \{(q, o', o, q), (q, \overline{o'}, \emptyset, q)\})$. Then, let the* join point program *of P and P_{JP}, noted $\boldsymbol{\mathcal{P}}(P, P_{JP})$, be defined as*

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

4.2.3.2 *toInit* and *toCurrent* Advice Weaving

A piece of *toInit* or *toCurrent* advice replaces the join point transitions by *advice transitions*: it redefines the target states and the outputs of the join point transitions, and optionally inserts an *advice program* before the new target states. The new target state of the advice transitions is defined by a finite input trace σ, which, for the *toInit* advice, is executed from the initial state, or, for the *toCurrent* advice, from the source state of the join point transition. The outputs O_{adv} of the advice transitions are given by the advice.

We first define weaving for *toInit* and *toCurrent* advice without advice programs in Definition 17, before giving the full definition of weaving in Definition 18, which also takes advice programs into account. Definition 17 is not strictly necessary and is not used in the remainder of the document, because it is generalized by Definition 18. However, it is easier to understand than Definition 18.

Weaving without Advice Programs. To make the understanding of the weaving process easier, Definition 17 defines a simplified form of weaving, not taking into account advice programs, i.e. we assume the last item of the advice to be $P_{adv} = (\{F\}, F, \emptyset, \emptyset, \emptyset)$. We first define the *targ* function, which determines the final state of an advice transition, and then define how weaving changes the transitions of the base program.

Definition 17 (*toInit*/*toCurrent* Advice Weaving without Advice Programs)**.** *Let* $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $adv = (type, O_{adv}, \sigma)$ *a piece of advice, with* $type \in \{toInit, toCurrent\}$, $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *and a finite input trace of length* $\ell_\sigma + 1$.

Let the target state determination function targ be defined as follows:

$$targ(s) = \begin{cases} S\_step_A(s_0, \sigma, \ell_\sigma) & \textit{iff } type = toInit \\ S\_step_A(s, \sigma, \ell_\sigma) & \textit{iff } type = toCurrent \end{cases} \tag{4.1}$$

The advice weaving operator without advice programs, $\triangleleft'_{JP}$*, weaves adv into A and returns the automaton* $A \triangleleft'_{JP} adv = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T}')$*, where* $\mathcal{T}'$ *is defined as follows:*

$$(s, \ell, O, s') \in \mathcal{T} \land JP \notin O \implies (s, \ell, O, s') \in \mathcal{T}' \tag{4.2}$$

$$(s, \ell, O, s') \in \mathcal{T} \land JP \in O \implies (s, \ell, O_{adv}, targ(s)) \in \mathcal{T}' \tag{4.3}$$

The *targ* function (defined by Equation (4.1)) determines the target state of an advice transition with source state s by executing σ either from the initial state, in case of a *toInit* advice, or from s, in case of a *toCurrent* advice. Therefore, S_step_A from Definition 7 (which has been naturally extended to finite input traces) executes the trace during ℓ_σ steps. Equations (4.2) and (4.3) define the changes to the transitions of A. Transitions (4.2) are not join point transitions and are copied unchanged from P. Transitions (4.3) emit *JP* and are thus replaced by the advice transitions, which have the same source state and condition, but O_{adv} as outputs and a new final state, given by *targ*.

Complete Weaving Definition. We now define the complete weaving algorithm for *toInit* and *toCurrent* advice. An aspect can contain an advice program P_{adv}, which is an Argos program with a special final state F. P_{adv} is executed after each advice transition, and F represents the re-entry point back to the program. The weaving inserts a copy of P_{adv} for every target state, and redirects the advice transitions to the initial state of the copy of P_{adv} corresponding to its target state. Furthermore, all the transitions that pointed to F in P_{adv} go to the corresponding target state in the copy.

Thus, in addition to the steps present in Definition 17, Definition 18 collects the target states in the base program, and creates a copy of P_{adv} for each target state it collected. The target states are collected into the set *TargSt*, and the states of the created copies in set *InsSt*. Furthermore, Definition 18 modifies the *targ* function, which determines the final state of the advice transitions. If an advice program is inserted, *targ* reconnects the advice transitions to the initial state of the corresponding copy of P_{adv}.

Definition 18 (*toInit*/*toCurrent* Advice Weaving)**.** *Let* $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $adv = (type, O_{adv}, \sigma, P_{adv})$ *a piece of advice, with* $type \in \{toInit, toCurrent\}$, $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *a finite input trace of length* $\ell_\sigma + 1$, *and* $P_{adv} = (\mathcal{Q}_{ins}, s_{0ins}, \mathcal{I}_{ins}, \mathcal{O}_{ins}, \mathcal{T}_{ins})$ *an advice program with final state* F.

Furthermore, let $TargSt = \{S\_step_A(s_0, \sigma, \ell_\sigma)\}$ *if* $type = toInit$ *or* $TargSt = \{s | \exists s' \in \mathcal{Q} \,.\, s = S\_step_A(s', \sigma, \ell_\sigma)\}$ *if* $type = toCurrent$ *be the set of all target states, and let* $InsSt = \{s_t | s \in \mathcal{Q}_{ins} \setminus \{F\}, t \in TargSt\}$ *be the set of the new states inserted by* P_{adv}. *Then, let the target state determination function targ be defined as follows:*

$$targ(s) = \begin{cases} S\_step_A(s_0, \sigma, \ell_\sigma) & \textit{iff } type = toInit \wedge s_{0ins} = F \\ S\_step_A(s, \sigma, \ell_\sigma) & \textit{iff } type = toCurrent \wedge s_{0ins} = F \\ (s_{0ins})_{S\_step_A(s_0, \sigma, \ell_\sigma)} & \textit{iff } type = toInit \wedge s_{0ins} \neq F \\ (s_{0ins})_{S\_step_A(s, \sigma, \ell_\sigma)} & \textit{iff } type = toCurrent \wedge s_{0ins} \neq F \end{cases} \tag{4.4}$$

The advice weaving operator, $\triangleleft_{JP}$, *weaves adv into A and returns the automaton* $A \triangleleft_{JP} adv = (\mathcal{Q} \cup InsSt, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T}')$, *where* $\mathcal{T}'$ *is defined as follows:*

$$(s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O \implies (s, \ell, O, s') \in \mathcal{T}' \tag{4.5}$$

$$(s, \ell, O, s') \in \mathcal{T} \wedge JP \in O \implies (s, \ell, O_{adv}, targ(s)) \in \mathcal{T}' \tag{4.6}$$

$$(s, \ell, O, s') \in \mathcal{T}_{ins} \wedge s' \neq F \wedge t \in TargSt \implies (s_t, \ell, O, s'_t) \in \mathcal{T}' \tag{4.7}$$

$$(s, \ell, O, F) \in \mathcal{T}_{ins} \wedge t \in TargSt \implies (s_t, \ell, O, t) \in \mathcal{T}' \tag{4.8}$$

TargSt is the set of all states that are selected by executing the trace σ. We insert a copy of P_{adv} before each of these. The states that are added to the woven program are created in *InsSt*, which contains a copy of each state of P_{adv} for each state in *TargSt*.

The target state determination function defined by Equation (4.4), $targ(s)$, computes the target state of an advice transition whose source is s. It distinguishes whether there is an advice program, and whether the type of the advice is *toInit* or *toCurrent*. If there is no advice program, it returns directly the state reached by executing the trace σ, as in Definition 17. If there is an advice program, *targ* does not return the state reached by executing σ, but the initial state of the copy of P_{adv} that has been inserted before it.

Equations (4.5) to (4.8) define the transitions of the woven program. Transitions (4.5) are not join point transitions and are copied unchanged from P. Transitions (4.6) emit *JP* and are thus replaced by the advice transitions, which have the same source state and condition, but O_{adv} as outputs and a new final state. The final state is given by the *targ* function (Equation (4.4)).

Transitions (4.7) are the transitions from the advice program that are copied for each instance. Transitions (4.8) lead from the advice program back into the base program, they replace the transitions that led to F in the advice program. They point to the state chosen by σ, before which their copy of P_{adv} was inserted.

Note that the complete weaving definition includes the weaving definition without advice programs: if $P_{adv} = (\{F\}, F, \emptyset, \emptyset, \emptyset)$, $A\triangleleft'_{JP}adv$ and $A\triangleleft_{JP}adv$ modify A syntactically in the same way, and then also $A\triangleleft'_{JP}adv \sim A\triangleleft_{JP}adv$.

4.2.3.3 Aspect Weaving for *toInit*/*toCurrent* Aspects

We now define the weaving of a *toInit* or a *toCurrent* aspect. It is woven into a program by first determining the join point transitions as defined in Definition 16 and then weaving the advice, as defined in Definition 18.

Definition 19 (*toInit*/*toCurrent* Weaving)**.** *Let P be a program and $asp = (P_{JP}, adv)$ an aspect for P. The weaving of asp on P is defined as follows:*

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

4.3 Recovery Aspects

This section introduces another kind of aspect, *recovery aspects*. In the first example, the pasteurizer in Section 4.2.1, aspect weaving modifies the program by adding transitions to a given point in the program, which is entirely specified 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. In Section 4.3.1, we first introduce the aspect informally on an example taken from the juice production department of the juice factory, and then give a formal definition in Section 4.3.2.

4.3.1 Example: the Blender

4.3.1.1 The Blender 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 first 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 round of production. Figure 4.10 shows the blender in its environment, and its interface is detailed in Figure 4.11.

4.3.1.2 Modifying the Blender Controller

The blender controller has the disadvantage that production has to be manually restarted each time the tank is full, and that the manifold reconnects each time a process restarts

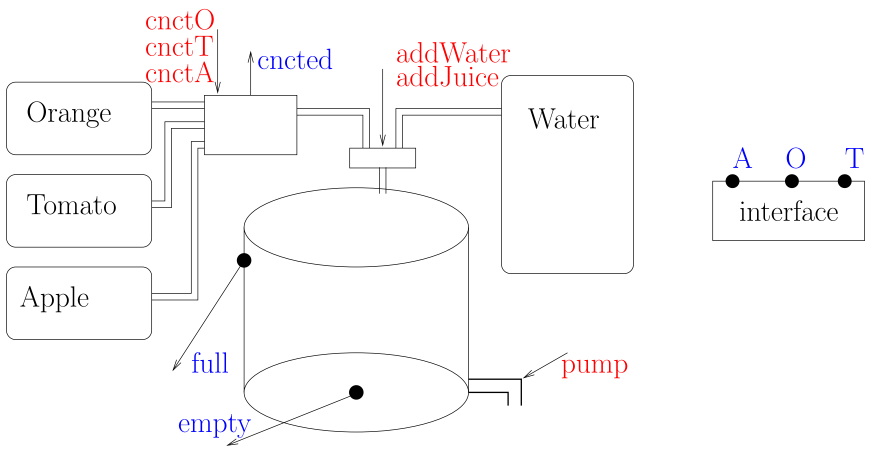


Figure 4.10: The blender and its environment.

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

Figure 4.11: The blender interface.

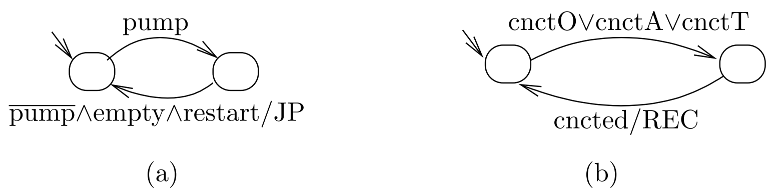

pump
$\overline{\text{pump}}\wedge$empty$\wedge$restart/JP
(a)
cnctO$\vee$cnctA$\vee$cnctT
cncted/REC
(b)

Figure 4.12: The pointcut (a) and the recovery program (b) for the recovery aspect.

(which 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 we want to restart the production of the same juice directly.

4.3.1.3 *recovery* Advice

We specify the additional functionality as follows: *"when* `restart` *is true immediately 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 select join point transitions and replace them with advice transitions.

Specifying join points. As before, we use a pointcut program to identify the join points transitions. To capture the condition *"immediately after the tank has been emptied"*, we can refer to the output `pump`, and wait until it switches from `true` to `false`, or we can refer to the sensor `empty`, and wait until it switches from `false` to `true`. We decide to use the output `pump`, because a sensor is more likely to show unpredicted behavior; e.g., it may send spurious signals while the tank is cleaned. We require, however, that the tank must also be empty, such that we do not capture cases where the controller just interrupted the emptying of the tank, e.g. because the next tank was full. Of course, the new input `restart` is also a condition for a join point. The pointcut that implements these requirements is shown in Figure 4.12(a).

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 in Figure 4.4, if we want to go backwards from the `Emptying` state by $\overline{\texttt{past}}$, 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 how to jump backwards. Some *recovery transitions* are defined globally and the program will only be able to return to the target state of the last recovery transition it has passed. To specify the recovery transitions, 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*. The transition in the program that is taken when *REC* is emitted is then selected as a recovery transition. We call the target states of the recovery transitions, to which the program jumps back to, *recovery states.*

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 in Figure 4.12(b).

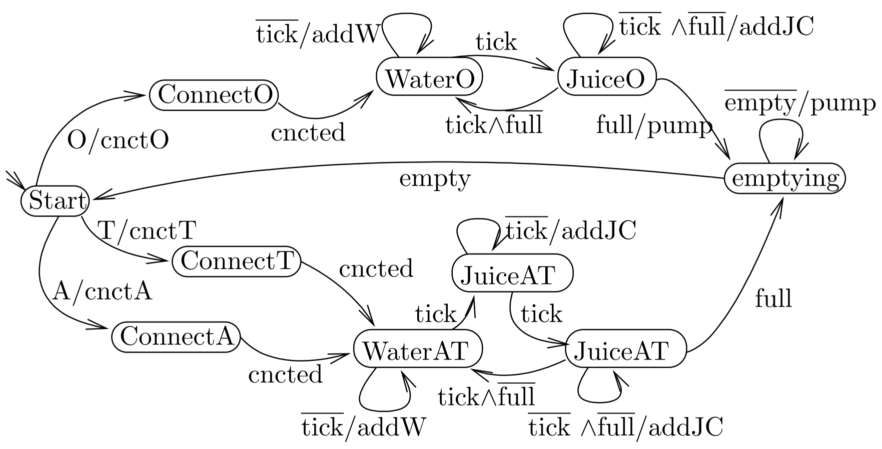

$\overline{\text{tick}}$/addW
tick
$\overline{\text{tick}}$ ∧$\overline{\text{full}}$/addJC
ConnectO
WaterO
JuiceO
cncted
tick∧full
full/pump
$\overline{\text{empty}}$/pump
O/cnctO
empty
emptying
Start
T/cnctT
$\overline{\text{tick}}$/addJC
ConnectT
cncted
JuiceAT
A/cnctA
tick
tick
full
ConnectA
WaterAT
JuiceAT
cncted
tick∧$\overline{\text{full}}$
$\overline{\text{tick}}$/addW
$\overline{\text{tick}}$ ∧$\overline{\text{full}}$/addJC

Figure 4.13: A sample blender controller.

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.

4.3.1.4 A Sample Blender Controller

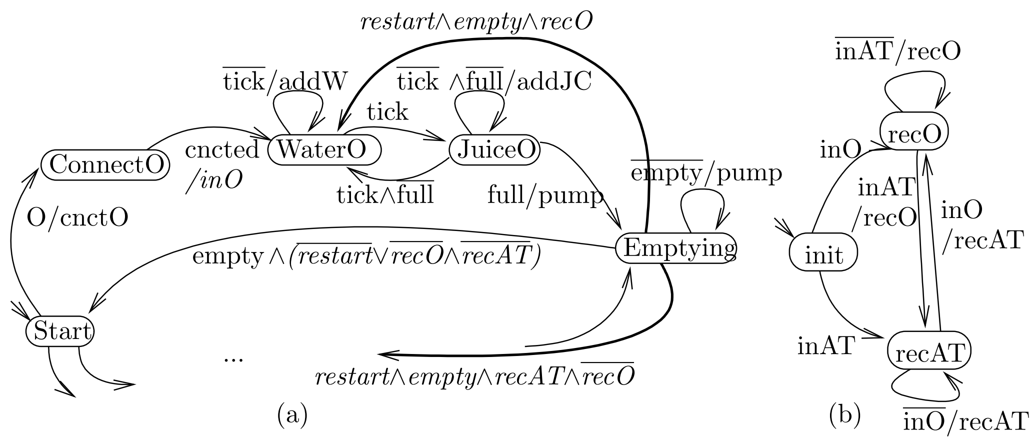
Figure 4.13 shows a sample implementation. From the `Start` state, a connection to the manifold is made. Then the juice is mixed using different proportions for different juices, one unit of water and one unit of juice for orange juice, one unit of water and two units of juice for tomato and apple juice. Finally, the content of the buffer is pumped to the next processing unit, before the controller goes back to state `Start`.

The only transition selected as join point transition is the one from state `Emptying` to state `Start`. It is split into two transitions, one where `restart` is false, and another one where `restart` is true. The latter is a join point transition.

The *recovery program* emits *REC* when `cncted` is emitted after the manifold has been asked to connect to some juice concentrate. This means that the aspect will go back in the controller to the point where it went after it received `cncted`, i.e. when it starts producing the juice. In the controller, it can either go back to state `WaterO`, if the last juice was orange juice, or to state `WaterAT`, if the last juice was apple or tomato juice. `WaterO` and `WaterAT` are thus the recovery states.

Once the join point transition and recovery states have been identified, advice transitions are added which replace the join point transition and which point to the recovery states. The upper half of the modified controller is displayed in Figure 4.14(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 a *memory automaton*, which emits the information to the modified controller, and which is run in parallel.

The memory automaton for the blender controller is shown in Figure 4.14(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 `WaterO` 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 passes a recovery transition, it emits either `inO` or `inAT`,



$restart \wedge empty \wedge recO$
$\overline{tick}$/addW
$\overline{tick}$ $\wedge\overline{full}$/addJC
tick
cncted
/inO
ConnectO
WaterO
JuiceO
tick∧full
full/pump
$\overline{empty}$/pump
O/cnctO
empty ∧ $(\overline{restart} \vee \overline{recO} \wedge \overline{recAT})$
Emptying
Start
...
$restart \wedge empty \wedge recAT \wedge \overline{recO}$
(a)
$\overline{inAT}$/recO
recO
inO
inAT
/recO
inO
/recAT
init
inAT
recAT
$\overline{inO}$/recAT
(b)

Figure 4.14: Result of weaving the blender aspect in the controller. Added states, transitions and modified conditions are written in italic.

depending on whether the transition led to `WaterO` or `WaterAT`. This makes the memory automaton update its state by entering the state `recO` or `recAT`. On the other hand, when the memory automaton is in its state `recO` (resp. `recAT`), 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 to.

The triggering condition of the advice transition to the recovery state `WaterO` is thus not only the activation signal `restart`∧`empty`, but `restart`∧`empty`∧`recO`, which indicates that `WaterO` was the last recovery state passed. A similar transition goes to `WaterAT`, with condition $\texttt{restart} \wedge \texttt{empty} \wedge \texttt{recAT} \wedge \overline{\texttt{recO}}$, where the additional $\overline{\texttt{recO}}$ is needed to keep the automaton deterministic. If $\overline{\texttt{recO}} \wedge \overline{\texttt{recAT}}$ is true, this means that no recovery state has been passed yet. In this case, the aspect does not modify the behavior of the controller; it does thus not modify the join point transition. The whole woven program controller is the parallel composition of the modified controller in Figure 4.14(a) and of the memory automaton in Figure 4.14(b), with an enclosing encapsulation of the signals `recO`, `recAT`, `inO`, and `inAT`.

4.3.2 Formal Definition

This section defines the weaving of *recovery* aspects formally. We first define *recovery* aspects, then the weaving of *recovery* advice, split into three parts, and finally the weaving of *recovery* aspects.

Definition 20 (*recovery* Aspect)**.** *A recovery aspect, for a program P on inputs $\mathcal{I}$ and outputs $\mathcal{O}$, is a tuple (P_{JP}, adv) where*

- $P_{JP} = (\mathcal{Q}_{pc}, s_{0pc}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{pc})$ *is the pointcut program, and JP occurs nowhere else in the environment.*
- $adv = (recovery, O_{adv}, P_{rec}, P_{adv})$ *is the advice, which contains four items:*
  - *recovery defines that this is an recovery aspect.*
  - $O_{adv} \subseteq \mathcal{O}$ *is the set of outputs emitted by the advice transitions.*

- $P_{rec} = (\mathcal{Q}_{rec}, s_{0rec}, \mathcal{I} \cup \mathcal{O}, \{REC\}, \mathcal{T}_{rec})$ *is the recovery automaton, which selects the recovery transitions, to which the aspect jumps back to.*
- $P_{adv} = (\mathcal{Q}_{ins} \cup \{F\}, s_{0ins}, \mathcal{I}_{ins}, \mathcal{O}_{ins}, \mathcal{T}_{ins})$, *with* $\mathcal{I}_{ins} \subseteq \mathcal{I}$ *and* $\mathcal{O}_{ins} \subseteq \mathcal{O}$, *is the advice program, which is executed after every advice transition. The final state F designates the return point to the program. If this item is not given, we assume it to be* $(\{F\}, F, \emptyset, \emptyset, \emptyset)$.

The selection of the join point transition is the same as for *toInit* and *toCurrent* advice, we re-use the operator $\boldsymbol{\mathcal{P}}(P, P_{JP})$ defined in Definition 16. We also use the same operator to select the recovery transitions.

4.3.2.1 *recovery* Advice Weaving

The recovery advice weaving is split into three parts: Definition 21 adds advice transitions to recovery states, Definition 22 adds the signals that inform the recovery automaton when a recovery state is reached, and Definition 23 defines the memory automaton, which remembers which recovery state was passed last. Definition 24 defines the weaving of recovery aspects, using the previous definitions.

Definition 21 (Recovery Transition Weaving)**.** *Let* $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton,* $adv = (recovery, O_{adv}, P_{rec}, P_{adv})$ *a recovery aspect for a program on* $\mathcal{I}$ *and* $\mathcal{O}$*, and let the set of recovery states* $\mathcal{R} = \{r_1, ..., r_n\} \subseteq \mathcal{Q}$ *be the states that have an incoming transition that emits REC. Let* $P_{adv} = (\mathcal{Q}_{ins}, s_{0ins}, \mathcal{I}_{ins}, \mathcal{O}_{ins}, \mathcal{T}_{ins})$*, and* $InsSt = \{s_t | s \in \mathcal{Q}_{ins} \setminus \{F\}, t \in \mathcal{R}\}$ *be the set of all states inserted by* P_{adv}*. The target state of the advice transition is then defined by*

$$targ^R(r_i) = \begin{cases} r_i & \textit{iff } s_{0ins} = F \\ (s_{0ins})_{r_i} & \textit{otherwise .} \end{cases} \quad \textit{for } r_i \in \mathcal{R}. \tag{4.9}$$

Furthermore, let $\mathcal{R}ec = \{rec_1, ..., rec_n\}$ *be fresh signals, and let the automaton* $A \triangleleft^R_{JP,\mathcal{R}} adv$ *be defined as* $(\mathcal{Q} \cup InsSt, s_0, \mathcal{I} \cup \mathcal{R}ec, \mathcal{O}, \mathcal{T}')$*, where* $\mathcal{T}'$ *is defined by*

$$\begin{array}{r} (s,\ell,O,s') \in \mathcal{T} \\ \wedge JP \in O \end{array} \Longrightarrow (s, \ell \wedge rec_i \wedge \bigwedge_{j=i+1..n} \overline{rec_j}, O_{adv}, targ^R(r_i)) \in \mathcal{T}' \tag{4.10}$$

$$\begin{array}{r} (s,\ell,O,s') \in \mathcal{T} \\ \wedge JP \in O \end{array} \Longrightarrow (s, \ell \wedge \bigwedge_{j=1..n} \overline{rec_j}, O \setminus \{JP\}, s') \in \mathcal{T}' \tag{4.11}$$

$$\begin{array}{r} (s,\ell,O,s') \in \mathcal{T} \\ \wedge JP \notin O \end{array} \Longrightarrow (s, \ell, O, s') \in \mathcal{T}' \tag{4.12}$$

$$\begin{array}{r} (s,\ell,O,s') \in \mathcal{T}_{ins} \\ \wedge s' \neq F \wedge t \in \mathcal{R} \end{array} \Longrightarrow (s_t, \ell, O, s'_t) \in \mathcal{T}' \tag{4.13}$$

$$\begin{array}{r} (s,\ell,O,F) \in \mathcal{T}_{ins} \\ \wedge t \in \mathcal{R} \end{array} \Longrightarrow (s_t, \ell, O, t) \in \mathcal{T}' \tag{4.14}$$

Transitions (4.10) are the advice transitions: every join point transition is replaced by transitions to all recovery states. The transition to state r_i is taken when the memory automaton $\mathcal{M}$ emits rec_i (meaning that the actual recovery state is r_i). We also add all negated

rec_j for $j > i$ to the condition, otherwise the automaton would not be deterministic if two rec_i were true at the same time. Transitions (4.11) treat the case when a join point transition is taken, but no recovery state has been passed yet (i.e., no *rec* signals are present). Then the original join point transition is left unmodified. Transitions (4.12) consider non join point transition, they are not modified.

Transitions (4.7) are the transitions from the advice program that are copied for each instance. Transitions (4.8) lead from the advice program back into the base program, they replace the transitions that led to F in the advice program. They point to the recovery state before which the corresponding copy of P_{adv} was inserted.

The next definition adds the $\mathcal{I}n$ signals to the program, which inform the memory automaton when a recovery transition is taken.

Definition 22 ($\mathcal{I}$ Recovery State Signals)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O} \cup \{REC\}, \mathcal{T})$ be an automaton, and $\mathcal{R} = \{r_1, ..., r_n\} \subseteq \mathcal{Q}$ be the states that have an incoming transition that emits REC. Then, $A \triangleleft^I_{\mathcal{R}} = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O} \cup \mathcal{I}n, \mathcal{T}')$, where $\mathcal{I}n = \{in_1, ..., in_n\}$ and $\mathcal{T}'$ is defined as follows:*

$$(s, \ell, O, r_i) \in \mathcal{T} \wedge REC \in O \implies (s, \ell, O \setminus \{REC\} \cup \{in_i\}, r_i) \in \mathcal{T}' \tag{4.15}$$

$$(s, \ell, O, s') \in \mathcal{T} \wedge REC \notin O \implies (s, \ell, O, s') \in \mathcal{T}' \tag{4.16}$$

Transitions (4.15) lead to recovery state r_i, and thus additionally emit in_i, such that the memory automaton can go to its corresponding state. Transitions (4.16) do not lead to a recovery state, they are left unchanged.

The following definition creates the memory automaton, and defines an operator that performs the parallel product with the program, and the encapsulation of the communication signals.

Definition 23 (Memory Automaton)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I} \cup \mathcal{R}ec, \mathcal{O} \cup \mathcal{I}n, \mathcal{T})$ be an automaton, $\mathcal{R}ec = \{rec_1, ..., rec_n\}$ and $\mathcal{I}n = \{in_1, ..., in_n\}$ signals, and $\mathcal{R} = \{r_1, ..., r_n\} \subseteq \mathcal{Q}$ a set of recovery states. The* memory automaton *$\mathcal{M}$ is then defined as $(\mathcal{Q}_\mathcal{M}, q_0, \mathcal{I}n, \mathcal{R}ec, \mathcal{T}_\mathcal{M})$, where $\mathcal{Q}_\mathcal{M} = \mathcal{R} \cup \{q_0\}$ and $\mathcal{T}_\mathcal{M}$ is defined by*

$$\begin{aligned} & \forall i \leq n \,.\, \forall j \leq n \,.\, (r_i, in_j \wedge \bigwedge_{j<k\leq n} \overline{in_k}, \{rec_i\}, r_j) \in \mathcal{T}_\mathcal{M} \\ \wedge \quad & \forall i \leq n \,.\, (r_i, \bigwedge_{1\leq j\leq n} \overline{in_j}, \{rec_i\}, r_i) \in \mathcal{T}_\mathcal{M} \\ \wedge \quad & \forall i \leq n \,.\, (q_0, in_i \wedge \bigwedge_{i<j\leq n} \overline{in_j}, \emptyset, r_i) \in \mathcal{T}_\mathcal{M} \,. \end{aligned} \tag{4.17}$$

Furthermore, $A \triangleleft^M_{\mathcal{R}}$ is defined as $(A \| \mathcal{M}) \setminus (\mathcal{R}ec \cup \mathcal{I}n)$.

4.3.2.2 Aspect Weaving for *recovery* Aspects

Finally, we put together Definitions 16, 21, 22, and 23 to define the weaving of recovery aspects.

Definition 24 (Recovery Weaving)**.** *The weaving of an recovery aspect $asp = (P_{JP}, adv = (recovery, O_{adv}, P_{rec}))$ into an automaton A, noted $A \triangleleft asp$, is defined as*

$$(\boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(P, P_{JP}), P_{rec}) \triangleleft^R_{JP,\mathcal{R}} adv) \triangleleft^I_{\mathcal{R}} \triangleleft^M_{\mathcal{R}} \,. \tag{4.18}$$

4.4 Conclusion

In this chapter, we have defined Larissa, an aspect language for Argos. The definition of aspects is given as a transformation of a flat automaton and an aspect into a flat automaton, in the same way as the other Argos operators. It respects one of the requirements given in Section 2.3.1 and in the introduction of this chapter, namely that it should be possible to syntactically combine aspect weaving freely with the other operators, and that the use of aspects should not be visible from the outside. The other requirements, the preservation of determinism and completeness, and of equivalence between base programs, are proven in Chapter 6.

Larissa selects transitions as join points, using a synchronous observer as a pointcut. Observers are a well-understood concepts in synchronous languages, that can express any safety property about a program. This property ranges only over the interface of the program, and it is completely independent of the implementation of the base program, thus respecting its encapsulation. Observers are easily understood by Argos programmers, because they are Argos programs themselves, and extensively used for testing and verification of synchronous programs.

Larissa has three kinds of advice, *toInit*, *toCurrent* and *recovery* advice. *toInit* advice allows to jump to a fixed point in the program, *toCurrent* advice to jump forward from the join point, and *recovery* backwards, to some statically selected recovery states. *toInit* and *toCurrent* advice specify the target states of the advice transitions by a finite trace over the inputs of the program, executed either from the initial state, or from the source state of the join point transition. This way of specifying the target state respects the encapsulation, because it does not refer to the implementation details, as would e.g. giving the name of the target state directly. The selection of the recovery states is made with observers, and thus has the same advantages as their use as a pointcut.

Furthermore, instead of jumping directly to a location in a program, Larissa can execute an advice program first, and go to the selected location afterwards. This considerably increases the expressive power of aspects: complete automata can be easily inserted, instead of just modifying transitions. As for the other parts of Larissa, advice programs are a familiar concept for the Argos programmer, and the way they are inserted respects the encapsulation of the base program.

The modifications performed by the aspects are quite general: we can select a set of transitions in an automaton, and then modify their target states and outputs, or insert a complete automaton at their place. The way we select transitions and target states are limited by our goal to respect the encapsulation of the program: we do not refer to the implementation of the base program, but only to its interface. We believe that, inside these limits, Larissa covers the ground of possible aspect extensions to Argos quite well, and can implement most cross-cutting concerns in Argos. However, further experimentation with Larissa will surely lead to modifications of the language. E.g., it may sometimes be useful to reuse the outputs from join point transition on the advice transition. It is also possible, as proposed by Shmuel Katz, to let an inserted program have several final states with different traces attached to them.

In this chapter, we used Larissa to encapsulate cross-cutting concerns in two examples, where it was used to add additional functionality to controllers in a juice processing plant. Because Larissa only refers to the interface of the base program, we could write the example aspects in this chapter without knowledge of the actual implementation of the base program.

This would not have been possible if the language referred to implementation details like state names. A larger example of Larissa is presented in Chapter 5.

Chapter 5

Case Study: Modelling the Interface of a Complex Wristwatch

5.1 Introduction

This chapter presents a case study, which uses Argos and Larissa to model the interface component of a complex wristwatch. The example is larger than the examples in the previous chapter, and it demonstrates the capability of Larissa to encapsulate cross-cutting concerns in an application for which Argos is a well-suited programming language. Furthermore, we show the value of using Larissa in the particular context of developing man-machine interfaces of small electronic devices.

The man-machine interface is a crucial component in a small electronic device like a wristwatch, as a great number of functions have to be controlled using a small set of buttons. These functions are ranged in a hierarchy of modes, and the buttons are used to traverse the modes and select the functions. In the implementation of such a device, the interface component is usually separated from the functional component, and is responsible for deciding the meaning of each button, depending on the current context.

Argos is well-suited to model such hierarchic interfaces, but some problems remain. One are cross-cutting concerns, which can not be encapsulated with the existing Argos operators. Furthermore, when building interfaces for related models, one would like to reuse components that have large parts in common, but differ in other parts. The latter parts often cannot be put in an Argos module of their own, and then we must maintain two slightly different versions of the same component. We use Larissa to address these problems: we encapsulate cross-cutting concerns, and assemble components such that we can build a product line of two watches by assembling smaller components, thus improving reuse.

This chapter is organized as follows: Section 5.2 presents the challenges of programming man-machine interfaces of small electronic devices, and Section 5.3 presents the case study, the implementation of a product line of wristwatches with Argos and Larissa. Section 5.4 presents some related work, and Section 5.5 concludes. The work presented in this chapter has been published in [AMS06b].

5.2 Modeling Interfaces of Small Devices

Man-Machine Interfaces of small electronic devices. Small electronic devices – e.g. wristwatches, alarm clocks or car radios – usually have a small number of buttons which control a large number of functionalities. For instance, the same button of a wristwatch means “toggle alarm” or “increment minutes”, depending on the running mode. These buttons have different meanings depending on the state in which the device is currently.

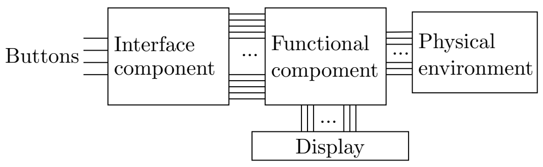


Figure 5.1: Typical structure of a small electronic device.

Therefore, controllers of such devices usually have a structure like the one shown in Figure 5.1: it contains an *interface component*, which interprets the meaning of the buttons the user presses, and then calls the corresponding function in the underlying *functional component*. The functional component has a much larger set of inputs (e.g. “toggle alarm” and “increment minutes”) than the interface component. The functional component also obtains the necessary information of the environment of the device (e.g. a quartz crystal to measure time), reads and writes persistent memory, and updates the display. This case study concentrates on the design of the *interface* component of such small electronic devices.

Programming Man-Machine Interfaces. Man-machine interfaces are typical interactive systems. Using reactive languages for programming or modeling them is quite natural, as reactive systems are a special case of interactive systems. Moreover, among the formalisms and languages that are used to describe reactive systems, those that are based on explicit automata, like Argos, are particularly well adapted. The user documentation of a small electronic device is often given with partial graphical automata, because it is the more natural way of thinking about it. However, some additional functions are difficult to express in a modular way.

Our case study shows two of these functions: additional modes and shortcuts. When a company proposes several variants of the same device, it is often the case that a large part of the internal components can be reused, provided the interface is modified to interpret new combinations of buttons, or new modes. A first kind of variant we consider consists of the addition of a new function, e.g. adding a barometer to a watch that already offers time-keeping functionalities. This often corresponds to the addition of a new “mode”, with all the related functions: set a particular value, reset, etc.

Another kind of variants consists in adding *shortcuts*. A shortcut is the possibility, in some given modes, to use a single button to activate a function that would otherwise need a long sequence of buttons. Adding shortcuts modifies the interface, but not the internal components.

5.3 Case Study: Suunto[1] Watches

As a case study, we implement the interface components of two wristwatches, the Altimax[1] and the Vector[1] models by Suunto[1]. Both have numerous (20 – 30) functionalities, which are controlled by a complex interface. Both watches share the same casing, display, and a large set of their functionalities: time, altimeter and barometer functions are nearly equal in both models, but the Vector also has an integrated compass. Carefully following the documentation [Suu], we propose Argos components and aspects to describe the interfaces of the two watches.

We structure the interfaces into the following components. A base program, written in Argos, contains the functionalities that both watches share. The behavior specific for one watch is added to the base program with aspects, including the aspects that model the shortcuts.

5.3.1 The Base Program

In both watches, each main functionality is represented by a main mode, which in turn has several submodes, that offer numerous functionalities. The interfaces of both watches contain four buttons, the `Mode`, the `Select`, the `Plus`, and the `Minus` button. The `Mode` button circles through the main modes, or, in a submode, returns to the main mode. The `Select` button selects a submode, and the `Plus` and `Minus` buttons modify current values. All main modes and many submodes have an associated configuration mode, where settings for the mode can be modified. A configuration mode can be reached by holding the select button pressed for two seconds in the corresponding mode. The interface then receives a special input, `s2s`.

Figure 5.2 shows the implementation of the interface component for the *base program*, i.e. the modes both wristwatches have in common. Each mode consists of a main mode, some submodes, and the associated configuration modes, that are reached through the input `s2s`. When the interface enters a (sub-)mode, it informs the functional component by emitting an output, such that it can update the display. Figure 5.2 is not complete: only some of the outputs (i.e., the commands to the functional component) are shown, namely `Time-Mode, Bari-Mode, Alti-Mode` and `mainMode`, which is emitted when a submode returns to its main mode. Furthermore, most of the states are further refined, and can emit commands, like `activateAlarm` in the Daily Alarms submode. Finally, the signal `toMainMode` is encapsulated: the submodes can emit it to force a return to their main mode. To save space, the encapsulation is not included in Figure 5.2.

5.3.2 The Altimax Watch

The first model we build, the Altimax, has the behavior of the base program, and a shortcut.

The Fast Cumulative Shortcut. The altimeter in the watches can record vertical movements in so called *logbooks*, so that the user can evaluate his performance after a hike. A logbook records the distances the user vertically ascents and descends from the moment it is started until it is stopped, and the number of runs accomplished in this period, i.e. the vertical movements of at least 50 meters. However, a logbook can only be read after recording stopped, and it is quite complicated to display the logbook (one has to go to the third submode of the altimeter main mode). Therefore, the Altimax model has the *fast cumulative*

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

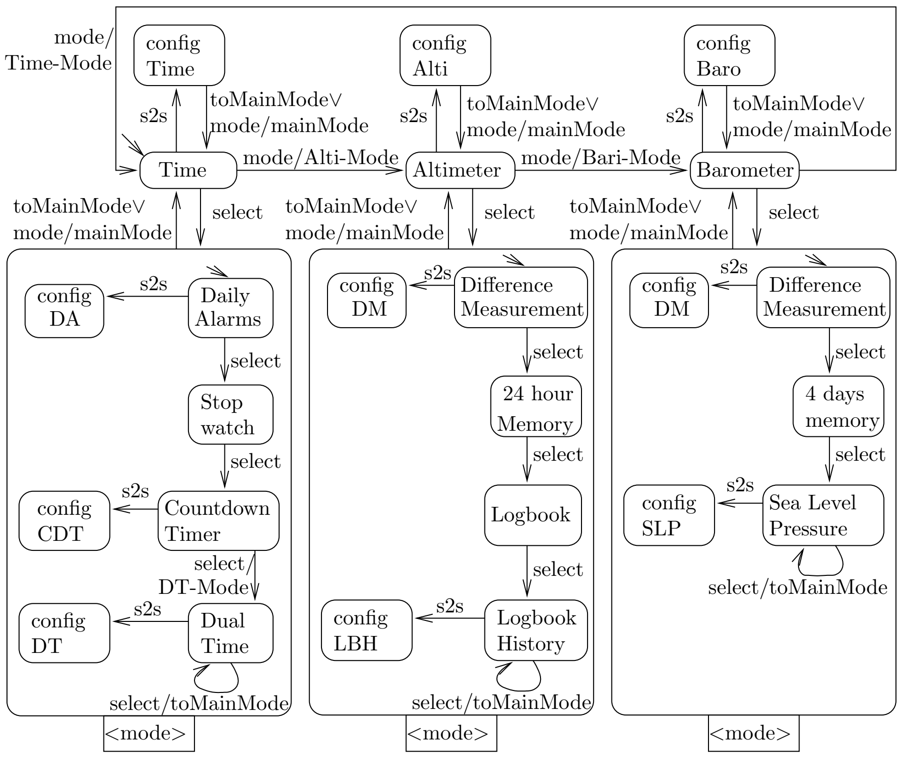


Figure 5.2: The `base-program` component. Its interface is: inputs = {`Mode, Select, plus, minus, s2s`}, outputs = {`Time-Mode, Bari-Mode, Alti-Mode, mainMode`}, `toMainMode` is encapsulated.

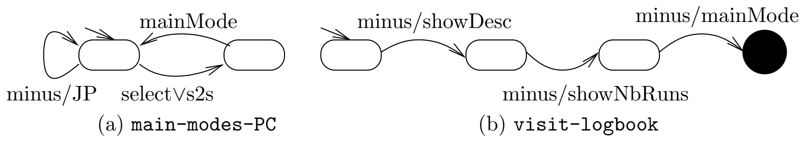
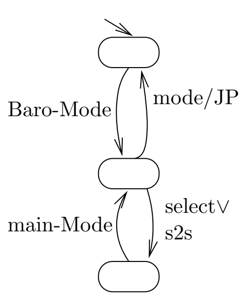


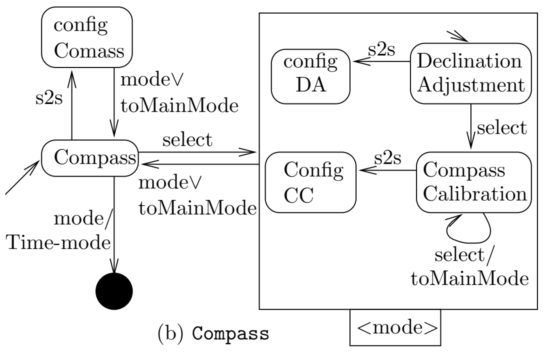
Figure 5.3: Pointcut (a) and advice program (b) for the Fast-Cumulative aspect.

shortcut: in any main mode, when the `Minus` button is pressed, some information from the current logbook is displayed. First the total vertical ascent rate is shown until the `Minus` button is pressed, then the total vertical descend rate and then the number of runs, before the watch returns to the main mode in which it was.

The fast cumulative mode is a typical shortcut and is implemented here with an aspect. The pointcut `main-modes-PC` in Figure 5.3 (a) chooses transitions which have a main mode of the base program as source state and `minus` as input part of the label. Visiting the current logbook is done in several steps: it first displays the ascent rate (output `showAsc`), then the descend rate (`showDesc`), and then the number of runs (`showNbRuns`). Therefore, the aspect emits first `showAsc` and then inserts the automaton `visit-logbook`, shown in Figure 5.3(b). As target state, we choose a trace of length zero (noted ϵ) from the current state, so that the program returns to the main mode in which it was when the aspect was activated. The aspect for the fast cumulative shortcut is fully specified by `Fast-Cumulative` = (`main-modes-PC`, (*toCurrent*, {`showAsc`}, ϵ, `visit-logbook`)).



(a) `baro-mode-PC`



(b) `Compass`

Figure 5.4: Pointcut (a) and advice program (b) for the `Compass-mode` aspect

Composing The Altimax Model. The controller of the Altimax watch is the base program (Figure 5.2) with the fast cumulative aspect woven to it: `Altimax` = `base-program`$\triangleleft$`Fast-Cumulative`.

5.3.3 The Vector Watch

We program a controller for the Vector wristwatch by applying three aspects to the base program, which are explained in the sequel.

The Compass Mode. The Vector has a fourth main mode, the compass mode. We add it to the base program with an aspect. The transition going from the Barometer main mode to the Time main mode is the sole join point transition (chosen by the pointcut `baro-mode-PC` in Figure 5.4 (a)). The only advice output is `Comp-Mode` which displays the compass. The aspect inserts the automaton `Compass` (see Figure 5.4 (b)), which contains the interface for the compass. After leaving the compass mode, the interface goes back to the `Time` main mode, thus the target state is set to the initial state: this is a *toInit* advice with σ being the empty trace ϵ. The resulting aspect is thus `Compass-Mode` = (`baro-mode-PC`, (*toCurrent*, {`Comp-Mode`}, ϵ, `Compass`)).

The Compass Shortcut. When the `Minus` button is pressed in a main mode, the Vector does not show information from the current logbook, but goes directly to the compass mode. This is useful when the user is hiking cross-country and wants to check regularly the bearing of the compass. Thus, the Vector does not contain the fast-cumulative aspect, but an aspect that adds advice transitions from the main modes to the compass main mode: `Fast-Compass` = (`main-modes-PC`, {`Comp-Mode`}, *toInit*, `mode.mode.mode.mode`). Note that this aspect must be applied after the `Compass-Mode` aspect, because it uses the compass mode. Indeed, after the compass mode has been added, it can be reached by pressing four times the `Mode` button from the initial state. The trace `mode.mode.mode.mode` ends in state `Compass`; it is the target state of the advice transitions.

No Dual Time Submode. As a last difference with the Altimax, the Vector lacks the Dual Time submode (the fourth submode of the `Time` main mode of the base program in Figure 5.2), which allows the user to simultaneously view the time in two different time zones. We cut it out of the base program with an aspect. We choose as join points all transitions which emit

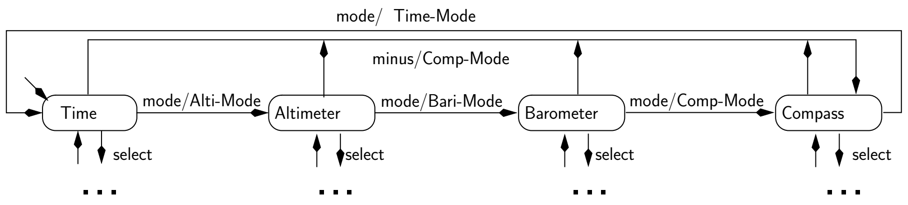


Figure 5.5: The woven interface of the Vector model.

DT-Mode, the signal that tells the underlying component to display the information related to the Dual Time mode. The corresponding pointcut, Countdown-PC, consists of a single state with a loop transition with label DT-Mode/JP. Instead of going to the Dual Time mode, the Vector goes to the Time main mode, thus the target state is defined by the empty trace. The aspect is thus defined by No-Dual-Time $=$ (Countdown-PC, (*toInit*, {Time-Mode}, ϵ)).

We could make the aspect more re-useable by defining the target state of the aspects as the following mode, reached by the trace select from the current state. Then, to remove another submode with this aspect, we just have to replace DT-Mode in the pointcut with the output of the submode we want to remove, and change the advice outputs (here Time-Mode) to the output of the next mode. The aspect is then defined as No-Dual-Time $=$ (Countdown-PC, (*toCurrent*, {Time-Mode}, select)).

Composing the Vector Model. The controller for the Vector can thus be built by weaving the three aspects into the base program: Vector $=$ base-program$\triangleleft$Compass-Mode$\triangleleft$Fast-Compass$\triangleleft$No-Dual-Time. Figure 5.5 shows a part of the automaton after the weaving of the Compass-Mode and Shortcut aspects. Only the main modes are shown, but not the sub- and configuration modes.

5.4 Related Work

We found very few papers which use AOP to build man-machine interfaces. [VH03] uses aspect-oriented programming to reduce the constraints imposed by the model-view-controller paradigm, which is central in many man-machine interfaces. The idea is that the type of software architecture imposed by man-machine interfaces (for instance because of the above-mentioned MVC paradigm) is sometimes very constraining, and aspects may be useful for reducing the constraints.

The use of aspects to build product lines is being studied extensively. We cite a few examples. Colyer and Clement [CC04] build a industrial product-line using AspectJ. Lopez-Herrejon and Batory [LHB05] propose a sequential weaving tool, similar to Larissa, to ease product-line development with aspects. Apel and Batory [AB06] combine Feature-Oriented Programming [BSR03], a technique to modularize product lines, with AOP. Loughran et al [LSR05] propose a tool to identify potential aspects early in the life-cycle of a product-line. This topic is also investigated in the EU AMPLE Project [AMP].

5.5 Conclusion

We have modeled the interface components of two complex wristwatches using Argos and Larissa. Larissa has been used to encapsulate shortcuts, and to build a product line of two watches. Shortcuts are typical cross-cutting concerns which cut across the different modes of the watches, and which cannot be expressed modularly by the Argos operators. With the Argos operators only, we also could not have built the product line without duplicating code. With Larissa, however, we could encapsulate into aspects the difference between the two watches, namely an additional main mode, a different shortcut, and a missing submode.

Thus, for a programming task Argos is very well-adapted to, we showed that Larissa is capable of expressing the cross-cutting concerns which appear in Argos programs. This indicates that Larissa can express cross-cutting concerns of typical applications written in Argos, and that it improves the expressiveness of Argos.

Furthermore, both shortcuts and small differences between similar models are typical for the man-machine interfaces of small electronic devices. The case-study indicates that Larissa is well-suited to encapsulate them, and thus that the development of such interfaces can profit from aspect-oriented languages like Larissa.

Chapter 6

Larissa Aspects as Argos Operators

6.1 Introduction

Larissa integrates best into Argos if aspect weaving can be considered another operator. Therefore, three conditions must be fulfilled: it must be possible to combine aspects freely with other operators, aspects must preserve the determinism and completeness of programs, and also preserve the equivalence between programs.

Because weaving is defined in the same way as the other operators, as a transformation into a flat automaton (see Definitions 18 and 24), it can be treated just as the other operators when building Argos programs, and thus fulfills the first criterion. Determinism and completeness are criteria which are important in the context of reactive systems. Although their preservation is not strictly necessary to be considered an operator (the encapsulation does not preserve them), it is very useful and should be preserved unless there is a good reason for not doing so.

The preservation of equivalence is the most restrictive of the above conditions. It means that the aspects must not refer to the implementation, but only to the interface of the program. Larissa follows these lines. First, it selects join points with a synchronous observer that only refers to the interface of the base program. The advice does not directly refer to the implementation either: the advice program and the advice outputs are independent of the implementation, and the return state is also selected through interface elements, a finite input trace in the case of *toInit* and *toCurrent* aspects, and an observer in the case of *recovery* aspects.

However, this alone does not guarantee the preservation of program equivalence: an aspect may refer to implementation details through interface elements, and thus distinguish between equivalent programs. E.g., this is the case if we select join point transitions with finite input traces. In this chapter, we prove that Larissa indeed preserves the equivalence, and does not indirectly distinguish between equivalent implementations.

Unlike Larissa, most aspect languages, notably AspectJ, do not preserve the equivalence. As an example, consider the two functions implementing a factorial in Figure 6.1. We consider them semantically equivalent, because they return the same result for any input, although they may have different execution times, and they are implemented differently.

```
int fact(int n) {
  int result = 1;
  while (n>1) {
    result = result * n;
    n--;
  }
  return result;
}

int fact(int n) {
  if (n>1)
    return n*fact(n-1);
  else
    return 1;
}

int around(int i): call(int *.*(int)) && args(i){
  return proceed(i-1);
}
```

Figure 6.1: Two equivalent implementations of factorial, and an aspect.

AspectJ Aspects can distinguish between the two implementations. Consider the aspect in the example in Figure 6.1, which subtracts 1 from every call to `fact`. After its application, the equivalent implementations of `fact` produce different results. This is because AspectJ aspects advise internal method calls, which are implementation details of the module.

AspectJ can break the encapsulation of programs in other ways: by advising private methods or fields, by constructs as `within`, which refer to the location of code, or by `privileged` aspects, which can access private members of a class in advice code. This problem has been recognized, and a number of solutions have been proposed. They are discussed in Section 10.4.

In Section 6.2 we prove the preservation of determinism and completeness, and in Section 6.3 the preservation of equivalence. In the proofs in this chapter, we will sometimes write $\triangleleft$ for $\triangleleft_{JP}$, $\triangleleft^R$ for $\triangleleft^R_{JP,\mathcal{R}}$ and $\triangleleft^I$ for $\triangleleft^I_{\mathcal{R}}$, where the meaning is clear from the context. Section 6.4 gives an alternative definition of weaving for a simple kind of aspects, which is defined directly on the traces of the base program. Defining weaving this way is only possible because aspects preserve equivalence between programs. Finally, Section 6.5 contains some concluding remarks. The theorems have been proved for a slightly different definition of aspects in [AMS06a].

6.2 Preservation of Determinism and Completeness

We first define a helper lemma (Lemma 1), and then prove the preservation of determinism and completeness in Theorem 1. Lemma 1 is used in the proof for Theorem 1, to show that the use of encapsulation in the definition of the aspect preserves determinism and completeness.

Lemma 1 identifies certain conditions that appear during the weaving process, and shows

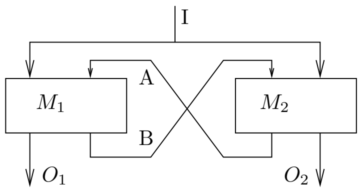


Figure 6.2: A dataflow diagram for two communicating automata, where A and B are encapsulated.

that under these conditions, the encapsulation of two communicating automata preserves determinism and completeness. The conditions describe a situation as depicted in Figure 6.2. Lemma 1 states that if the communication signals A emitted by M_2 in instant t do not depend on the communication signals B emitted by M_1 in instant t, no cyclic dependencies can form.

Lemma 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 a state* s *of* M_2 *emit the same subset of* A, *i.e.* $\forall t_1 = (s_1, \ell_1, o_1, s_1') \in \mathcal{T}_2, t_2 = (s_2, \ell_2, o_2, s_2') \in \mathcal{T}_2 \;.\; s_1 = s_2 \Rightarrow o_1 \cap A = o_2 \cap A$. *Then,* $M_1 \| M_2 \setminus (A \cup B)$ *is complete and deterministic.*

Proof. All outgoing transitions of 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 note $\tilde{a}$ 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 $M_1 \| M_2$, 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 $M_1 \| M_2 \setminus (A \cup B)$. 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 $\ell_1' \wedge \ell_1 = \texttt{false}$, we have $\ell_1' \wedge \tilde{a} = \texttt{false}$, 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 with source state 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 $\ell_2' \wedge \tilde{b} = \texttt{false}$ and the same reasoning applies, such that the transition is cut. □

We now show that aspect weaving preserves both completeness and determinism. Using Lemma 1, we show that the encapsulations in the definition of weaving preserves completeness and determinism, and that the modifications of transitions in the definition of advice weaving do the same.

Theorem 1 (Preservation of determinism and completeness)**.** *Let P be a program and $asp = (P_{JP}, adv)$ an aspect. Let P, P_{JP}, and, if present in the advice, also the advice program (except in F) and the recovery program, be deterministic and complete. Then, $P \lhd asp$ is also deterministic and complete.*

6.2.1 Proof for Theorem 1

We prove that weaving preserves determinism and completeness. We prove the theorem for programs being simple automata, since weaving first flattens a program it is applied to. Weaving an aspect is defined using the parallel product, the encapsulation and advice weaving. The parallel product preserves determinism and completeness. We will show that advice weaving does so, too. The encapsulation does not always preserve determinism nor completeness. However, we will show that the particular cases in which encapsulation is used for weaving also preserve both.

***toInit*/*toCurrent* aspects.** Let A be an automaton, and let $asp = (P_{JP}, (type, O_{adv}, \sigma, P_{adv}))$, with $type \in \{toInit, toCurrent\}$. We prove that if A, P_{JP} and P_{adv} (with the exception of F) are deterministic and complete, then so is $A \lhd asp$.

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

Second, weaving the advice into $\boldsymbol{\mathcal{P}}(A, P_{JP})$ preserves determinism and completeness: some transitions are left unchanged, and some have their outputs and target states modified. This does not alter determinism or completeness of the affected state. Furthermore, all new states are complete and deterministic, too: F has disappeared, and we required the other states of P_{adv} to be complete and deterministic, as are the states to which σ points, because they were reachable in $\boldsymbol{\mathcal{P}}(A, P_{JP})$, which is deterministic and complete.

Recovery aspects. Let A be an automaton, and let $asp = (P_{JP}, (recovery, O_{adv}, P_{rec}, P_{adv}))$. We prove that if A, P_{JP}, P_{rec} and P_{adv} (with the exception of F) are deterministic and complete, then so is $A \lhd asp$. The proof follows the same steps as above.

First, as for *toInit* aspects, $\boldsymbol{\mathcal{P}}(A, P_{JP})$ is deterministic and complete. The same reasoning also applies to $P = \boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A, P_{JP})P_{rec})$.

Second, we show that applying Definition 21 preserves determinism and completeness. Non join point transitions only have their outputs modified, which does not influence determinism or completeness. A join point transitions with condition ℓ is replaced by different transitions: one with $\ell \wedge \bigwedge_{1 \leq i \leq n} \overline{rec}_i$, and for every $i \leq n$ one with condition $\ell \wedge rec_i \wedge \bigwedge_{i < j \leq n} \overline{rec}_j$. These conditions are pairwise disjoint, the automaton is thus deterministic. Their disjunction is ℓ, thus the automaton is complete. Hence $P \lhd^R adv$ is deterministic and complete. Furthermore, $\lhd^I$ only changes outputs, and thus does not affect determinism and completeness.

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 $in_j \wedge \bigwedge_{j<k\leq n} \overline{in_k}$ 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, $A \triangleleft asp$ is obtained by $(\boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A, P_{JP}), P_{rec}) \triangleleft^R adv \triangleleft^I \| \mathcal{M}) \setminus (\mathcal{R}ec \cup \mathcal{I}n)$. We have shown that $\boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A, P_{JP}), P_{rec}) \triangleleft^R adv \triangleleft^I$ and $\mathcal{M}$ are deterministic and complete, thus so is their product. To show that the encapsulation does not affect determinism nor completeness, we apply again Lemma 1 : we set $M_1 = \boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A, P_{JP}), P_{rec}) \triangleleft^R adv \triangleleft^I$, $M_2 = \mathcal{M}$, $A = \mathcal{R}ec$ and $B = \mathcal{I}n$. The $\mathcal{R}ec$ outputs of $\mathcal{M}$ only depend on the current state, it always emits rec_i when in state r_i or $\emptyset$ when in state q_0. Thus the condition Lemma 1 imposes on M_2 are fulfilled. $\square$

6.3 Preservation of Equivalence

Theorem 2 shows that the trace equivalence (see Definition 9) is preserved by aspect weaving.

Theorem 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, $P_1 \sim P_2 \implies (P_1 \triangleleft asp) \sim (P_2 \triangleleft asp)$.*

6.3.1 Proof for Theorem 2

We prove the preservation of semantic equivalence for *toInit*, *toCurrent*, and *recovery* aspects. We prove the theorem for programs being simple automata, since weaving effectively operates on such automata once the Argos operators have been applied.

We first introduce a definition of equivalence between states and transitions.

Definition 25 (State and Transition Equivalence). *Let A_1 and A_2 be two automata and let q_1 be a state of A_1 and q_2 a state of A_2. q_1 and q_2 are said to be* equivalent *(noted $q_1 \sim q_2$) iff $A_1' \sim A_2'$ where A_1' (resp. A_2') is the automaton A_1 (resp. A_2) where the initial state is set to q_1 (resp. q_2).*

Furthermore, two transitions (q_1, ℓ, O, q_1') and (q_2, ℓ, O, q_2') are equivalent *iff $q_1 \sim q_2$ and $q_1' \sim q_2'$.*

Note that the definition of state equivalence only considers the future execution from the states, but not how they can be reached. We will need this kind of equivalence in the proof.

We first prove the theorem for *toInit* and *toCurrent* aspects, and then for *recovery* aspects.

***toInit/toCurrent* aspects.** Let A_1 and A_2 be automata, and let $asp = (P_{JP}, (type, O_{adv}, \sigma, P_{adv}))$, with $type \in \{toInit, toCurrent\}$, and P_{adv} optional. We prove that if A_1 and A_2 are semantically equivalent, then $A_1 \triangleleft asp$ and $A_2 \triangleleft asp$ are also semantically equivalent.

$A_1 \sim A_2 \implies \boldsymbol{\mathcal{P}}(A_1, P_{JP}) \sim \boldsymbol{\mathcal{P}}(A_2, P_{JP})$, because $\boldsymbol{\mathcal{P}}$ only relies on operators $\|$ and $\setminus$, which are known to preserve equivalence (see [Mar92] for a proof). Furthermore, iff a transition in $\boldsymbol{\mathcal{P}}(A_1, P_{JP})$ is a join point transition, then so are all equivalent transitions in $\boldsymbol{\mathcal{P}}(A_2, P_{JP})$, otherwise they would be distinguishable by *JP*.

If *asp* is applied to two equivalent join point transitions, they are modified in the same way. They emit O_{adv}, and the target states selected by σ are also equivalent; otherwise the starting states of the trace would be distinguishable. If $type = toInit$, the traces start in the initial states of A_1 and A_2, which are equivalent because of $A_1 \sim A_2 \implies \boldsymbol{\mathcal{P}}(A_1, P_{JP}) \sim \boldsymbol{\mathcal{P}}(A_2, P_{JP})$.

If $type = toCurrent$, the traces start in the initial states of the join point transitions, which are also equivalent, because they were reachable by the same trace. Trivially, adding P_{adv} also affects A_1 and A_2 in the same way. Thus, $A_1 \triangleleft asp \sim A_2 \triangleleft asp$.

Recovery aspects. Let A_1 and A_2 be automata, and let $asp = (P_{JP}, adv))$, with $adv = (recovery, O_{adv}, P_{rec}, P_{adv})$ and $P_{JP} = (\mathcal{Q}_{\text{pc}}, s_{0\text{pc}}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{\text{pc}})$. We have already shown that $A_1 \sim A_2 \implies \boldsymbol{\mathcal{P}}(A_1, P_{JP}) \sim \boldsymbol{\mathcal{P}}(A_2, P_{JP})$. We obtain $\boldsymbol{\mathcal{P}}(A_1, P_{JP}) \sim \boldsymbol{\mathcal{P}}(A_2, P_{JP}) \implies \boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A_1, P_{JP}), P_{rec}) \sim \boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A_2, P_{JP}), P_{rec})$ by exactly the same reasoning. We denote $P'_i = \boldsymbol{\mathcal{P}}(\boldsymbol{\mathcal{P}}(A_i, P_{JP}), P_{rec})$ for $i = 1 \ldots 2$. $P'_1 \triangleleft^R adv$ and $P'_2 \triangleleft^R adv$ are not equivalent (they may have even different in- and outputs: the $\mathcal{I}n$ and $\mathcal{R}ec$ signals), nor are $\mathcal{M}_1$ and $\mathcal{M}_2$. However, we show that $(P'_1 \triangleleft^R adv \| \mathcal{M}_1) \setminus (\mathcal{R}ec_2 \cup \mathcal{I}n_1)$ and $(P'_2 \triangleleft^R adv \| \mathcal{M}_2) \setminus (\mathcal{R}ec_2 \cup \mathcal{I}n_2)$ are trace equivalent.

Let (it, ot) be a trace of $A_1 \triangleleft asp$. By induction over the length of the trace, we show that (it, ot) is also a trace of $A_2 \triangleleft asp$.

Inductive hypothesis:

$$\forall m \leq n \ . \ S\_step_{A_1 \triangleleft asp}(s_{01}, it, m) \sim_{\overline{JP}} S\_step_{A_2 \triangleleft asp}(s_{02}, it, m)$$

where the conditional equivalence $\sim_{\overline{JP}}$ is defined by

$$X \sim_{\overline{JP}} Y \Leftrightarrow \big[(\forall (it, ot) \ . \ (\forall n \ . \ O\_step_{P_{JP}}(s_{0\text{pc}}, it.ot, n+1) = \emptyset) \\ \Rightarrow ((it, ot) \in Traces(X) \Leftrightarrow (it, ot) \in Traces(Y)))\big] \ .$$

The conditional equivalence $\sim_{\overline{JP}}$ denotes trace equivalence for all traces where the aspect is never activated, because the pointcut never emits JP.

Base Case: For $n = 0$, S_step returns the initial state. We must thus show $A_1 \triangleleft asp \sim_{\overline{JP}} A_2 \triangleleft asp$. When JP is false, only those transitions in $P'_i \triangleleft^R adv$ are taken that have not been modified with respect to the transitions in P'_i, except for the addition of $\mathcal{I}n$ signals. The transitions in $P'_i \triangleleft^R adv$ that take the signals $\mathcal{R}ec$ into account have all been selected by the pointcut, and do thus not influence $\sim_{\overline{JP}}$. Thus, the memory automaton has no effect on $P'_i \triangleleft^R adv$, because it only communicates through $\mathcal{R}ec$. Thus, $A_i \triangleleft asp \sim_{\overline{JP}} P'_i$ and because of $P'_1 \sim P'_2$, we have $A_1 \triangleleft asp \sim_{\overline{JP}} A_2 \triangleleft asp$.

Inductive step: We show that if the hypothesis is true for n, it is also true for $n+1$, and we also show $O\_step_{A_1 \triangleleft asp}(s_{01}, it, n+1) = O\_step_{A_2 \triangleleft asp}(s_{02}, it, n+1))$. We distinguish two cases: either (1) P_{JP} 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_{adv}, we have thus $O\_step_{A_1 \triangleleft asp}(s_{01}, it, n+1) = O\_step_{A_2 \triangleleft asp}(s_{02}, it, n+1))$. Let $t_r < n$ be the last instant in $1 \ldots n-1$ where REC is emitted, and let r_1 and r_2 be the recovery states passed in t_r. 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, $P'_1 \triangleleft^R adv$ (resp. $P'_2 \triangleleft^R adv$) takes the advice transition leading to r_1 (resp. r_2). Because of the induction hypothesis we have $r_1 \sim_{\overline{JP}} r_2$.

(2) Only those transitions in $P'_i \triangleleft^R adv$ are taken that existed already in P'_i. Because of $P'_1 \sim P'_2$ and the induction hypothesis, we have $S\_step_{A_1 \triangleleft asp}(s_{01}, it, n+1) \sim_{\overline{JP}} S\_step_{A_2 \triangleleft asp}(s_{02}, it, n+1)$ and also $O\_step_{A_1 \triangleleft asp}(s_{01}, it, n+1) = O\_step_{A_2 \triangleleft asp}(s_{02}, it, n+1))$.

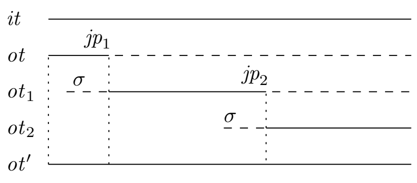


Figure 6.3: Illustration of the trace weaving defined in Definition 26. (it, ot) is the original trace, the ot_i are the outputs of traces starting with σ, and ot' is the output part of a trace of the woven program. The jp_i are the join points.

Because we inductively showed that $O\_step_{A_1 \triangleleft asp}(s_{01}, it, n) = O\_step_{A_2 \triangleleft asp}(s_{02}, it, n))$ holds for any n, $A_1 \triangleleft asp$ and $A_2 \triangleleft asp$ have the same outputs for it, thus (it, ot) is also a trace of $A_2 \triangleleft asp$. □

6.4 Trace Transformation Semantics

Because the semantics of aspect weaving is independent of the implementation of the base program, it can also be defined directly on the trace semantics of the base program, without referring to its implementing automaton. This has the advantage of giving the semantics of weaving explicitly on the semantics of the program, i.e. its set of traces.

It is, however, less intuitive and more complicated than defining aspects on automata. We thus give such a semantics only for *toInit* aspects, with no advice programs. The definition for this simple kind of aspect is already quite complicated. Given that automata are also formally defined, we restrain ourselves from defining the complete Larissa language this way.

The trace transformation semantics is based on a concatenation of chunks of traces from the base program. The first chunk starts at the beginning of the trace, and ends when the pointcut emits *JP* for the first time. The following chunks are all taken from traces that start with the inputs from the input trace σ. The part from the end of σ until the next occurrence of *JP* is then used to build the trace of the woven program. This process is illustrated in Figure 6.3.

Definition 26 (Trace Weaving). *Let Traces*(P) *be the set of traces of an deterministic and complete automaton* P, σ *a finite input trace of length* ℓ_σ, *and let* $asp = (P_{JP}, (toInit, O_{adv}, \sigma))$ *be an aspect. We define the* trace weaving *of asp into Traces*(P), *noted Traces*$(P) \triangleleft asp$.

Let $(it, ot) \in Traces(P)$. *Then,* $(it, ot') \in Traces(P) \triangleleft asp$, *where* ot' *is defined by*

$$ot'(i) = \begin{cases} ot(i) & \text{if } i < jp_1 \text{ or if } jp_1 \text{ does not exist.} \\ ot_n(i - jp_n + \ell_\sigma) & \text{if } jp_n < i < jp_{n+1} \\ & \text{or if } jp_n < i \text{ and } jp_{n+1} \text{ does not exist.} \\ O_{adv} & \text{iff } i = jp_n, \end{cases} \tag{6.1}$$

where jp_1, *the first join point identified by* P_{JP} *in* (it, ot), *is defined by*

$$\exists jp_1 \,.\, O\_step_{P_{JP}}(s_{0P_{JP}}, it.ot, jp_1) = \{JP\} \\ \wedge \forall i < jp_1 \,.\, O\_step_{P_{JP}}(s_{0P_{JP}}, it.ot, i) = \emptyset, \tag{6.2}$$

it_n is defined by

$$it_n(i) = \begin{cases} \sigma(i) & \textit{iff } i \leq \ell_\sigma \\ it(jp_n + i - \ell_\sigma) & \textit{iff } i > \ell_\sigma, \end{cases} \tag{6.3}$$

ot_n is defined by

$$\exists ot_n \,.\, (it_n, ot_n) \in \mathit{Traces}(P), \tag{6.4}$$

and jp_i for all $i > 1$ is defined by

$$\begin{aligned} &\exists jp_{n+1} > jp_n \,.\, O\_step_{P_{JP}}(s_{0P_{JP}}, it_n.ot_n, jp_{n+1} - jp_n + \ell_\sigma) = \{JP\} \\ &\qquad \wedge \forall \ell_\sigma < i < jp_{n+1} - jp_n + \ell_\sigma \,.\, O\_step_{P_{JP}}(s_{0P_{JP}}, it_n.ot_n, i) = \emptyset \,. \end{aligned} \tag{6.5}$$

We prove that the trace weaving has the same semantics as the automata weaving defined in Definition 18. This is another way of proving the preservation of trace equivalence by the weaving operator: if the semantics is defined on the set of traces, the weaving for trace equivalent automata results in the same set of traces. However, the direct proof of trace equivalence is simpler, we thus use it to prove the preservation of trace equivalence for the complete language.

Theorem 3 (Correctness of Trace Weaving)**.** *Let P be an automaton, and let $asp = (P_{JP}, (toInit, O_{adv}, \sigma))$ be an aspect. Then, we have*

$$\mathit{Traces}(P \triangleleft asp) = \mathit{Traces}(P) \triangleleft asp \,.$$

Proof. $\mathit{Traces}(P \triangleleft asp)$ and $\mathit{Traces}(P) \triangleleft asp$ are both deterministic and complete, $\mathit{Traces}(P \triangleleft asp)$ by Theorem 1, and $\mathit{Traces}(P) \triangleleft asp$ because Definition 26 defines exactly one trace for each trace of P, which is deterministic and complete.

Thus, because we know that there is exactly one pair of i/o-traces for every input trace in both $\mathit{Traces}(P) \triangleleft asp$ and $\mathit{Traces}(P \triangleleft asp)$, it is sufficient to show

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

Let $(it, ot) \in \mathit{Traces}(P)$, consistent with Definition 26. Proof by induction on the application of advice to it.

Induction hypothesis : At step jp_n, $P \triangleleft asp$ takes an advice transition.

Base case: By Equation (6.2), jp_1 is the first time P_{JP} emits JP for (it, ot), and thus also the first time $\boldsymbol{\mathcal{P}}(P, P_{JP})$ emits JP. Therefore, $P \triangleleft asp$ also has trace (it, ot) until jp_1, and takes an advice transition at jp_1.

By the first case of Equation (6.1), (it, ot') is also equal to (it, ot) until jp_1. Thus, $\forall i < jp_1, ot'(i) = ot(i) = ot''(i)$.

Induction step: We show that if the induction hypothesis is true at jp_n, it is also true at jp_{n+1}, and that ot' and ot'' have the same outputs between jp_n and jp_{n+1}.

At jp_n, $P \triangleleft asp$ takes an advice transition by the induction hypothesis. Thus, $ot''(jp_n) = O_{adv}$ by the definition of weaving (Definition 18 in Chapter 4). We also have $ot'(jp_n) = O_{adv}$ by the third case of Equation (6.1).

Furthermore, by Definition 18, $P \triangleleft asp$ goes to $S\_step_{\boldsymbol{\mathcal{P}}(P,P_{JP})}(s_0, \sigma, \ell_\sigma)$ at jp_n, the state reached after executing σ on $\boldsymbol{\mathcal{P}}(P, P_{JP})$.

By the definition of weaving, when $P \triangleleft asp$ and $\boldsymbol{\mathcal{P}}(P, P_{JP})$ are leaving from the same state, both produce the same trace until $\boldsymbol{\mathcal{P}}(P, P_{JP})$ emits JP the next time. Trace it_n in Equation (6.3) concatenates σ and the part of it following jp_n. Let jp'_{n+1} be the next time after jp_n that $P \triangleleft asp$ takes an advice transition in (it, ot''). If it_n is applied to $\boldsymbol{\mathcal{P}}(P, P_{JP})$, between ℓ_σ and jp'_{n+1}, it produces the same outputs as $P \triangleleft asp$ after jp_n, and it emits JP for the next time (after ℓ_σ) at $jp'_{n+1} - jp_n + \ell_\sigma$, i.e. when $P \triangleleft asp$ takes an advice transition. Formally, we know that

$$\begin{aligned} &JP \in O\_step_{\boldsymbol{\mathcal{P}}(P,P_{JP})}(s_{0P_{JP}}, it_n, jp'_{n+1} - jp_n + \ell_\sigma) \\ &\qquad \wedge \forall \ell_\sigma < i < jp'_{n+1} - jp_n + \ell_\sigma \;.\; JP \notin O\_step_{\boldsymbol{\mathcal{P}}(P,P_{JP})}(s_{0P_{JP}}, it_n, i)\;. \end{aligned} \tag{6.6}$$

Furthermore, P and $\boldsymbol{\mathcal{P}}(P, P_{JP})$ emit the same traces except for JP, and ot_n is the output trace produced by P for it_n. Then, by definition of $\boldsymbol{\mathcal{P}}$, P_{JP} emits JP for the trace $it_n.ot_n$ iff $\boldsymbol{\mathcal{P}}$ emits JP for it. We can thus rewrite Equation 6.6 into

$$\begin{aligned} &JP \in O\_step_{P_{JP}}(s_{0P_{JP}}, it_n.ot_n, jp'_{n+1} - jp_n + \ell_\sigma) \\ &\qquad \wedge \forall \ell_\sigma < i < jp'_{n+1} - jp_n + \ell_\sigma \;.\; JP \notin O\_step_{P_{JP}}(s_{0P_{JP}}, it_n.ot_n, i)\;. \end{aligned} \tag{6.7}$$

By comparing Equation (6.7) to Equation (6.5), we see that jp'_{n+1} and jp_{n+1} denote the same steps, and thus the induction hypothesis is true for $n+1$, because $P \triangleleft adv$ takes an advice transition at jp_{n+1}.

Finally, between jp_n and jp_{n+1}, $ot'(i) = ot_n(i - jp_n + \ell_\sigma)$ by the second case of Equation (6.1), and is thus equal to $O\_step_{P \triangleleft asp}(s_{0P_{JP}}, it_n, i)$. □

6.5 Conclusion

In this chapter, we have shown that aspect weaving preserves determinism and completeness of programs, and equivalence between programs. Furthermore, by their definition in Chapter 4, aspects can be combined freely with other operators. Thus, aspect weaving can be considered as another operator of Argos.

Therefore, we can give a new Grammar for Argos and Larissa, and add aspect weaving as an additional operator. Programs are now expressions built from single automata with synchronous product, encapsulation, inhibition, refinement, and weaving of aspects.

Definition 27 (Grammar of Argos and Larissa)**.** *The set of Argos expressions is defined by the grammar:*

$$\begin{array}{rll} P ::= & P \| P & \textit{parallel product} \\ | & P \setminus \Gamma & \textit{encapsulation} \\ | & P \; \texttt{whennot} \; a & \textit{inhibition} \\ | & A \triangleright \{R_i\}_{i=0\ldots n} & \textit{refinement} \\ | & P \triangleleft asp & \textit{aspect weaving} \\ R ::= & P \quad | \quad \texttt{NIL} & \textit{refining objects,} \end{array}$$

where A is an automaton as defined in Definition 6, Γ a set of signals, a a signal, and asp an aspect. `NIL` *represents a state that is not refined.*

Furthermore, we have defined an alternative way to define the semantics of weaving for *toInit* aspects without advice programs. Instead of letting an aspect transform an automaton, it directly transforms its semantics, specified as a set of traces. This way of specifying is much more complicated and difficult to understand, however. We therefore do not define the rest of Larissa this way.

Chapter 7

Interference between Aspects

7.1 Introduction

In aspect-oriented programs, more than one aspect is often being applied to a program. These aspects can interact in different ways, and the order in which the aspects are applied may depend on this interaction. E.g., in the wristwatch example from Chapter 5, the compass shortcut aspect clearly depends on the aspect that adds the compass mode, and can only be applied if the compass mode aspect has already been woven. In this case, the interaction between is thus desired by the programmer.

On the other hand, aspects are often designed independently of each other, and are not meant to interact. In this case, their interaction may lead to unintended behavior of the woven program, and we say that the aspects interfere. The interference between aspects is a major problem in aspect-oriented programming. This has been recognized by many authors (e.g. [DFS02, DFS04]), and much work has been devoted to detecting and resolving aspect interferences, which we will discuss in Section 7.6. In this chapter, we discuss this problem for Larissa aspects.

Formally, we say that aspects that should not interact *interfere* if applying them in a different order yields different results. If A_1 and A_2 are aspects, and weaving first A_1 and then A_2 yields a different program than weaving first A_2 and then A_1, A_1 and A_2 are said to interfere.

If two aspects interfere may depend on the way they are woven. In this chapter, we will specifically compare two different weaving definitions, that greatly influence aspect interference. The approach consists in weaving aspects *sequentially*, i.e. one by one after one another. Larissa weaves aspects sequentially: an aspect is applied to an automaton, and the next aspect is applied to the resulting automaton, without taking into account that some of the behavior comes from an aspect. This corresponds to the operator model of Argos, where the operators are defined as automata transformers. Thus, the weaving of two aspects A_1 and A_2 into a program P is a two step process, which we note $(P \triangleleft A_1) \triangleleft A_2$.

The second approach consists in weaving aspects *jointly*, i.e. by applying all the aspects together, at the same time to the same base program. Weaving A_1 and A_2 then cannot be truly separated into two independent steps, and we note it $P \triangleleft \{A_1, A_2\}$. E.g., AspectJ weaves aspects jointly. It defines the semantics of an aspect not as a transformation of the base program, but rather as injecting behavior in the execution of the *woven* program, including the execution of other aspects. During weaving, aspects must thus know about each other. Both aspects influence each other, as opposed to sequential weaving, where only the second aspect influences the first one.

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

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

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

Figure 7.1: A small Java program and two AspectJ aspects.

In general, sequential weaving often causes interference. As an explanation, let us look at the example in Figure 7.1. The class `Test` has a method `foo` and a method `main`, which calls `foo` on a `Test` object. Furthermore, there are two aspects. Aspect `A1` has a method `bar()` and adds a call to `bar` at the end of every call to every method named `foo`. After compiling together the class `Test` and the aspect `A1` (`Test`◁`A1`), the execution of `main` executes `foo()` and then `bar()`.

The second aspect, `A2`, adds some code at the end of every method named `bar`. If the class `Test` is compiled with `A2` only (`Test`◁`A2`), nothing changes. The class `Test` is unchanged, since no call to method `bar` exists, we have `Test` = `Test`◁`A2`.

Imagine that a weaver for AspectJ produces Java code as a backend, and that for weaving two aspects, it first weaves the first one, obtains some Java code, and weaves the second aspect into the result. This weaver applies a sequential weaving strategy, the same as Larissa.

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

In real AspectJ, aspects are of course woven jointly. In the example, this produces the same result as (`Test`◁`A1`)◁`A2`.

In this chapter we propose a weaving mechanism for Larissa that weaves several aspects jointly into a program, and thus eliminates some cases of interference. As opposed to AspectJ, pointcuts do not capture join points in the woven program, but in the base program only. These different kinds of weaving have been discussed first in [DFS02], where the kind that captures join points in the base program only is termed *silent*, and the other kind *visible.*

Using a silent weaving algorithm has the disadvantages that joint weaving in Larissa cannot replace sequential weaving. We still need to weave aspects sequentially in some cases, when the second aspects must be applied to the result of the first. E.g., consider the compass aspect from Chapter 5, which adds an additional main mode to the watch, and the compass-shortcut aspect, which adds transitions to that main mode. The shortcut aspect and the compass aspect cannot be woven jointly, because the shortcut aspect needs to refer to the states that the compass aspect introduced. It must thus be sequentially woven *after* the compass aspect, so that it can select the new main mode as a target state.

```
before() : execution(* *.a()){b();}
before() : execution(* *.b()){a();}
```

Figure 7.2: Two AspectJ aspects which will result in a stack overflow if either method `a` or `b` is called.

On the other hand, silent weaving also has advantages. It makes the definition of weaving easier, and it avoids the problem that aspects which recursively call each other may lead to non-termination. Figure 7.2 contains an example of such aspects in AspectJ. In the context of Larissa, weaving join points in advice may lead to a similar problem if two aspects create advice transitions that are selected as join point transitions by the other aspect. This would likely lead to a non-terminating weaving algorithm.

Introducing non-terminating aspects in AspectJ is not a big problem, because it is also easy to build non-terminating programs with Java alone. However, Larissa aspects that may lead to an undefined woven program or non-terminating weaving process are not acceptable.

Although they are jointly woven, aspects in AspectJ may still interfere. This is illustrated by the example in Figure 7.3. The sets of join points selected by `A3` and by `A4` are the same. The interference here is unavoidable since the advice programs have to be executed sequentially. In such a case, AspectJ allows to describe the order of application of advice with the `declare precedence` keyword, as in Line 2 in Figure 7.3.

```
1 aspect A3 {
2   declare precedence : A4, A3;
3   before(): call(* foo (..)) { ... }
4 }
5 aspect A4 {
6   before(): call(* foo (..)) { ... }
7 }
```

Figure 7.3: Two interfering AspectJ aspects.

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

We do not study interference for sequentially woven aspects, and do not show how non-interference can be proven in this case. This would be much more difficult, and it makes no sense: if two aspects are meant not to interfere, they should be jointly woven. In cases where we need sequential weaving, i.e. when one aspect depends on the other, interference is unavoidable, because the aspect that depends on the other must be woven last.

The remainder of this chapter is structured as follows: Section 7.2 introduces an example, which is used in Section 7.3 to illustrate aspect interference in Larissa and introduce joint weaving. Section 7.4 then describes a way to formally prove non-interference, and Section 7.3.1 extends this approach to *recovery* aspects. Section 7.5 contains the proofs for the theorems introduced in the previous sections. Finally, Section 7.6 explores some related work and Section 7.7 concludes. The work presented in this chapter has partly been published in [SAM06] and [Sta07a].

7.2 Example

We re-use the wristwatch example from Chapter 5 in this section, but adapt it to produce interesting cases of interference. Therefore, we apply two new shortcut aspects to the same watch. The shortcut aspects are not those from the real watches presented in Chapter 5, but simpler shortcuts, that just jump to a submode.

Consider the Altimax watch, with the base program shown in Figure 5.2. The `plus` and the `minus` buttons have both no function consistent with their intended meaning in the main modes, we can thus use both as a shortcut. We introduce one shortcut that jumps to the logbook of the altimeter, activated by `minus`, and one that jumps to the four day memory of the barometer, activated by `plus`. We term the aspect that introduces the first shortcut the *logbook aspect*, and the one that introduces the second the *memory aspect*. We can re-use the pointcut from the fast cumulative aspect in Figure 5.3 for one aspect, and must modify the other such that it emits *JP* when `plus` is pressed in a main mode. These pointcuts are shown in Figure 7.4. The two pointcuts emit different join point signals, JP_l and JP_m, for the logbook and the memory aspect respectively.

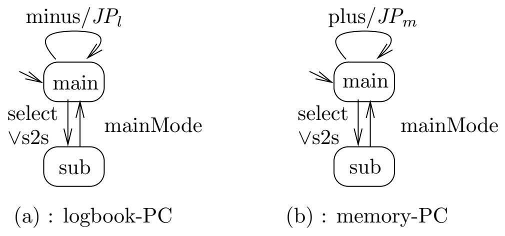


Figure 7.4: The pointcuts for the shortcut aspects.

As advice, we specify the trace that leads to the functionality we want to reach, i.e. σ_l = `mode.select.select.select` for the logbook aspect and σ_m = `mode.mode.select.select` for the 4-day memory aspect, and the output that tells the underlying component to display the corresponding information. Thus, we use the aspects $logbook - asp$ = (`logbook-PC`, ($toInit$, {`showLogbook`}, σ_l)) and $memory - asp$ = (`memory-PC`, ($toInit$, {`showBaroMemory`}, σ_m)).

7.3 Interfering Aspects

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

The problem is that the memory aspect has been written for the program without the logbook aspect: the pointcut assumes that the only way to leave a main mode is to press the `select` or the `s2s` button. However, the logbook aspect invalidates that assumption by adding transitions with condition `minus` from the main modes to a submode. When these transitions are taken, the pointcut of the memory aspect incorrectly assumes that the program is still in a main mode.

Furthermore, for the same reason, applying first the memory aspect and then the logbook aspect produces (in terms of trace-equivalence) a different program from applying first the logbook aspect and then the memory aspect: `watch`$\triangleleft$`logbook-asp`$\triangleleft$`memory-asp` $\not\sim$ `watch`$\triangleleft$`memory-asp`$\triangleleft$`logbook-asp`.

As a first attempt to define *aspect interference*, we say that two aspects A_1 and A_2 interfere when their application on a program P in different orders does not yield two trace-equivalent programs: $P \triangleleft A_1 \triangleleft A_2 \not\sim P \triangleleft A_2 \triangleleft A_1$. We say that two aspects that do not interfere are *independent*.

With interfering aspects, the aspect that is woven second must know about the aspect that was applied first. To be able to write aspects as the ones above independently from each other, we propose a mechanism to weave several aspects jointly. The idea is to first determine the join point transitions for all the aspects, and then apply the advice. We first define joint weaving only for *toInit*/*toCurrent* aspects, which is easier to understand than the full definition that also takes *recovery* aspects into account, and is sufficient for the example used in this chapter. We define jointly weaving all kinds of aspects in Definition 29.

Definition 28 (Joint weaving of several *toInit*/*toCurrent* aspects)**.** *Let $A_1, \ldots, A_n$ be some toInit or toCurrent aspects, with $A_i = (P_{JP_i}, adv_i)$, and P a program. We define the application of $A_1, \ldots, A_n$ on P as follows:*

$$P \triangleleft (A_1, \ldots, A_n) = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n}) \triangleleft_{JP_n} adv_n \ldots \triangleleft_{JP_1} adv_1 \ .$$

Note that Definition 28 reuses the advice weaving operator defined in Definition 18, and indexes the join point signal used by each advice. Furthermore, the advice is woven in the reverse order, i.e. we first the advice from the last aspect in the aspect list, and the advice from the first aspect last. This way, aspects that are later in the list have higher priority: if a join point transition is claimed by several aspects, the one that is woven first replaces the join point transition with its advice transition, and removes the join point signals of the other aspects. To give priority to the aspects that are applied later is consistent with sequential weaving, where aspects that are applied later modify the aspects that have already been applied, but not the other way round.

Jointly weaving the logbook and the memory aspect leads to the intended behavior, because both aspects first select their join point transitions in the main modes, and change the target states of the join point transitions only afterwards.

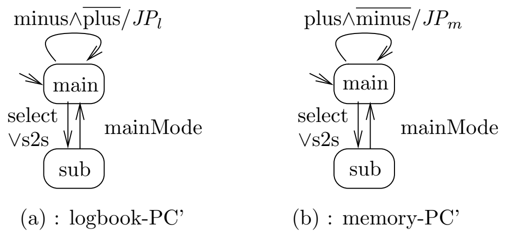


Figure 7.5: Two non interfering pointcuts for the shortcut aspects.

However, the order in which the aspects are woven still influences the program. The two aspects share some join point transitions, namely when both buttons are pressed at the same time in a main mode. Both aspects then want to execute their advice, but only one can. Only the aspect that was applied last is executed. Thus the two aspects interfere with each other. However, this problem can be solved by modifying the pointcuts, such that they do not share any join point transitions. E.g., we can add $\overline{\texttt{plus}}$ to the join point transition of the logbook aspect's pointcut and $\overline{\texttt{minus}}$ to the join point transition of the memory aspect's pointcut, as shown in Figure 7.5. With these pointcuts, the order of weaving the two aspects does not influence the final program.

7.3.1 Extension to *recovery* Aspects

Above, we defined joint weaving for *toInit* and *toCurrent* aspects only. When jointly weaving *recovery* aspects, it is not sufficient to just separate the join point weaving, as we do in Definition 28. If we first weave the complete *recovery* advice of the first aspect, i.e. the three operators $\triangleleft^R$, $\triangleleft^I$, and $\triangleleft^M$ defined in Definitions 21, 22, and 23 respectively, then the complete advice of the second and so on, we have the same problem as with sequential weaving.

Consider two *recovery* aspects R_1 and R_2, with R_1 woven after R_2. Each time R_1 takes an advice transition, R_2's memory automaton is reset to the state in which it was when R_1 last passed a recovery state, because R_1 recovers the *whole* automaton it was applied to, including R_2's memory automaton. On the other hand, when R_2 takes an advice transition, this does not influence the memory automaton of R_1, because it is added later. Furthermore, if we weave first R_1 and then R_2, this is the other way round. Thus, even if R_1 and R_2 have no join point transitions in common, they may interfere.

Therefore, we define a generalized version of joint weaving, which applies the $\triangleleft^I$ and $\triangleleft^M$ operators after $\triangleleft^R$ and $\triangleleft_{JP}$ operators.

Definition 29 (Joint weaving of several aspects)**.** *Let* $\mathcal{A} = A_1 \dots A_n$ *be some aspects, with* $A_i = (P_{JP_i}, adv_i)$*,* $\mathcal{A}_R = \{A_{R1} \dots A_{Rm}\} \subseteq \mathcal{A}$ *the recovery aspects in* $\mathcal{A}$*, with* $adv_i = (recovery, O_{advi}, P_{reci}, P_{advi})$ *if* $A_i \in \mathcal{A}_R$*,* P *a program, and let* $P_{pc} = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n} \| P_{recR1} \| \dots \| P_{recRm})$*. We define the application of* $A_1 \dots A_n$ *on* P *as follows:*

$$P \triangleleft (A_1, \dots, A_n) = P_{pc} \triangleleft^?_n adv_n \dots \triangleleft^?_1 adv_1 \triangleleft^I_{\mathcal{R}_{Rm}} \dots \triangleleft^I_{\mathcal{R}_{R1}} \triangleleft^M_{\mathcal{R}_{Rm}} \dots \triangleleft^M_{\mathcal{R}_{R1}}$$

where $\triangleleft^?_i = \triangleleft^R_{JP_i, \mathcal{R}_i}$ *if* $A_i \in \mathcal{A}_R$ *and* $\triangleleft^?_i = \triangleleft_{JP_i}$ *otherwise.*

Definition 29 solves the above problem: as when weaving *toInit*/*toCurrent* aspects, aspects do not influence each other except by removing each others join point signals. Especially, all memory automata are not influenced by other advice.

7.3.2 Defining Interference

Formally, we define aspect interference for the application of several aspects as follows, using the extended definition that takes all kinds of aspects into account.

Definition 30 (Aspect Interference)**.** *Let A_1 ... A_n be some aspects, and P a program. We say that A_i and A_{i+1}* interfere *for P iff*

$$P \triangleleft (A_1, \ldots, A_i, A_{i+1}, \ldots, A_n) \nsim P \triangleleft (A_1, \ldots, A_{i+1}, A_i, \ldots, A_n)$$

When two jointly woven aspects interfere, as do aspects with the pointcuts shown in Figure 7.4, the conflict should be made explicit to the programmer, so that it can be solved by hand, as we did for the example above. A mechanism to identify such conflicts is introduced in the following section.

7.4 Proving Non-Interference

In this section, we show that in some cases, non-interference of aspects can be proven, if the aspects are woven jointly, as defined in Definition 29. We can prove non-interference of two given aspects either for any program the aspects are woven into, or for two aspects and a given program. Following [DFS02, DFS04], we speak of *strong independence* in the first case, and of *weak independence* in the second.

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

Definition 31 (*jpTrans*)**.** *Let $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be an automaton and $JP \in \mathcal{O}$. Then,*

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

The following theorem proves strong independence between two aspects, i.e. it proves that two aspects do not interfere for any base program.

Theorem 4 (Strong Independence)**.** *Let $A_1 \ldots A_n$ be some aspects, with $A_i = (P_{JP_i}, adv_i)$, and let P be a program. Then, the following equation holds:*

$$\begin{aligned}&\mathit{jpTrans}(P_{JP_i} \| P_{JP_{i+1}}, JP_i) \cap \mathit{jpTrans}(P_{JP_i} \| P_{JP_{i+1}}, JP_{i+1}) = \emptyset \\ &\Rightarrow P \triangleleft (A_1 \ldots A_i, A_{i+1} \ldots A_n) \sim P \triangleleft (A_1 \ldots A_{i-1}, A_{i+1}, A_i, A_{i+2} \ldots A_n) \ .\end{aligned}$$

A proof is given in Section 7.3.1. Theorem 4 states that if there is no transition with both JP_i and JP_{i+1} as outputs in the product of P_{JP_i} and $P_{JP_{i+1}}$, A_i and A_{i+1} are independent and thus can commute while weaving their advice. Theorem 4 defines a sufficient condition for non-interference, by looking only at the pointcuts. When the condition holds, the aspects are said to be *strongly independent.*

Theorem 5 (Weak Independence)**.** *Let $A_1 \ldots A_n$ be some aspects, with $A_i = (P_{JP_i}, adv_i)$, P a program, and $P_{pc} = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \ldots \| P_{JP_n})$. Then, the following equation holds:*

$$\begin{aligned}&\mathit{jpTrans}(P_{pc}, JP_i) \cap \mathit{jpTrans}(P_{pc}, JP_{i+1}) = \emptyset \\ &\Rightarrow P \triangleleft (A_1, \ldots, A_i, A_{i+1}, \ldots, A_n) \sim P \triangleleft (A_1, \ldots, A_{i+1}, A_i, \ldots, A_n) \ .\end{aligned}$$

A proof is given in Section 7.3.1. Theorem 5 states that if there is no transition with both JP_i and JP_{i+1} as outputs in P_{pc}, A_i and A_{i+1} do not interfere. This is weaker than Theorem 4 since it also takes the program P into account. However, there are cases in which the condition of Theorem 4 is false (thus it yields no results), but Theorem 5 allows to prove non-interference. Section 7.4.2 contains such an example.

Note that both Theorem 4 and Theorem 5 only prove independence for aspects that are adjacent in the weaving list. By applying the theorem repeatedly, we can also prove independence for two non-adjacent aspects, if both aspects are independent of the aspects between them.

Theorem 5 is a sufficient condition, but, as Theorem 4, it is not necessary: it may not be able to prove independence for two independent aspects. One reason is that it does not take into account the effect of the advice weaving: consider two aspects such that the only reason why the condition for Theorem 5 is false is a transition with source state s, such that s is only reachable through another join point transition; if the advice weaving makes this state unreachable, then the aspects do not interfere. Another possibility for an incorrectly detected conflict is the case where the two pieces of advice have the same effect on the program: then, the weaving order obviously does not matter, even if the aspects share join point transitions.

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

Note that the detection of non-interference is a static condition that does not add any complexity overhead. Indeed, to weave the aspects, the compiler needs to build first $P_{JP_1} \|$ ... $\| P_{JP_n} = P_{\text{all}JP}$: the condition of Theorem 4 can be checked during the construction of $P_{\text{all}JP}$. Second, the weaver builds $P_{\text{pc}} = \boldsymbol{\mathcal{P}}(P, P_{\text{all}JP})$, and it can check the condition of Theorem 5. Thus, to calculate the conditions of both theorems, it is sufficient to check the outputs of the transitions of intermediate products during the weaving. The weaver can easily emit a warning when a potential conflict is detected.

To have an exact characterization of non-interference, it is still possible to compute the predicate $P \triangleleft (A_1 \dots A_i, A_{i+1} \dots A_n) \sim P \triangleleft (A_1 \dots A_{i+1}, A_i \dots A_n)$, but calculating semantic equality is very expensive for large programs.

Note that the interference presented here only applies to the joint weaving of several aspects, as defined in Definition 28. Sequentially woven aspects may interfere even if their join points are disjoint, because the pointcut of the second aspects applies to the woven program. A similar analysis to prove non-interference of sequential weaving would be more difficult, because the effect of the advice must be taken into account. Indeed, the advice of an aspect influences which transitions are selected by the pointcut of an aspect that is sequentially woven next.

7.4.1 Interference between the Shortcut Aspects

Let us apply the formal interference analysis to the example from Section 7.2. Figure 7.6 (a) shows the product of the modified pointcuts of the logbook and the memory aspect from Figure 7.5. There are no transitions that emit both JP_l and JP_m, thus, by applying Theorem 4, we know that the aspects do not interfere, independently of the program they are applied to.

Consider again that the original pointcuts of the logbook and the memory aspect from

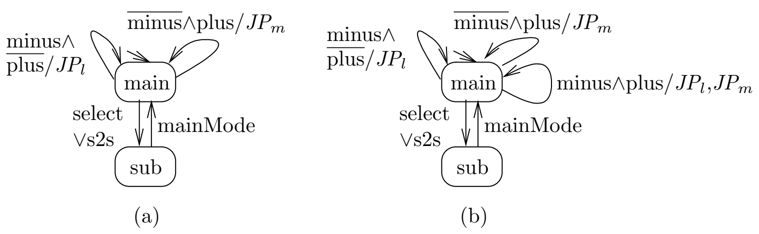


Figure 7.6: Interference between shortcut pointcuts.

Figure 7.4. The product of the two pointcuts is shown in Figure 7.6 (b). State `main` has another loop transition, with label `minus∧plus/`JP_l`,`JP_m. Thus, Theorem 4 not only states that the aspects potentially interfere, but it also states precisely where the interference may occur: here, the problem is that when both `minus` and `plus` are pressed in a main mode, at the same time, both aspects are activated. A compiler for Larissa can thus emits a warning, and the programmer can solve the conflict if needed.

7.4.2 Interference between a Shortcut and the No-DTM Aspect

We now apply the formal interference analysis to another example taken from the watch in Chapter 5. Consider the interference between shortcut aspect and the No-DTM aspect from Section 5.3.3, which removes the Dual Time submode from the watch.

The pointcut of the No-DTM aspects emits the join point signal JP_n, and is a single-state automaton which emits JP_n when the input `DT-Mode`, which activates the Dual Time mode, is true. Figure 7.7 shows the initial state of the product of the pointcuts of the logbook (Figure 7.4 (a)) and the No-DTM aspect. There is a transition that has both JP_l and JP_n as outputs. Theorem 4 states that the aspects may interfere, but when applied to the wristwatch controller, they do not. This is because the `DT-Mode` is an output of the controller and is never emitted when the watch is in a main mode, where the logbook aspect can be activated. As the `DT-Mode` is always false in the main modes, the conflicting transition is never enabled. When applied to another program, however, the aspects may interfere.

In this example, the use of Theorem 5 is thus needed to show that the aspects do not interfere when applied to the wristwatch controller. Its condition is true, as expected, because JP_l is only emitted in the main modes, and JP_n only in the Time submodes.

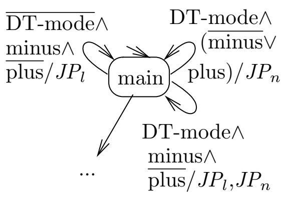


Figure 7.7: Interference between a shortcut and No-DTM pointcuts.

7.5 Proofs

In this section, we prove Theorems 4 and 5, using the joint weaving defined in Definition 29, which also takes the complete Larissa language into account.

We first prove a lemma we will need in the proof for Theorem 5, then Theorem 5, and finally Theorem 4, which is a direct consequence of Theorem 5.

Lemma 2 (Distributivity of the Encapsulation). *Let* $P_1 = (\mathcal{Q}_1, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1)$ *and* $P_2 = (\mathcal{Q}_2, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2)$ *be automata, and* Γ *signals such that* $\Gamma \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$. *Then, we have*

$$(P_1 \setminus \Gamma) \| P_2 \sim (P_1 \| P_2) \setminus \Gamma \; .$$

Proof. By applying first Definition 11 and then Definition 10, we obtain $(P_1 \setminus \Gamma) \| P_2 = (\mathcal{Q}_1 \times \mathcal{Q}_2, \mathcal{I}_1 \setminus \Gamma \cup \mathcal{I}_2, \mathcal{O}_1 \setminus \Gamma \cup \mathcal{O}_2, \mathcal{T}')$, where $\mathcal{T}'$ is defined by

$$\begin{aligned}(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \;\wedge\; \ell_1^+ \cap \Gamma \subseteq O_1 \;\wedge\; \ell_1^- \cap \Gamma \cap O_1 = \emptyset \;\wedge\; (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \\ \iff (s_1 s_2, \exists \Gamma \,.\, \ell_1 \wedge \ell_2, O_1 \setminus \Gamma \cup O_2, s_1' s_2') \in \mathcal{T}' \; .\end{aligned}$$

On the other hand, by applying first Definition 10 and then Definition 11, we have $(P_1 \| P_2) \setminus \Gamma = (\mathcal{Q}_1 \times \mathcal{Q}_2, (\mathcal{I}_1 \cup \mathcal{I}_2) \setminus \Gamma, (\mathcal{O}_1 \cup \mathcal{O}_2) \setminus \Gamma, \mathcal{T}'')$, where $\mathcal{T}''$ is defined by

$$\begin{aligned}&(s_1, \ell_1, O_1, s_1') \in \mathcal{T}_1 \;\wedge\; (s_2, \ell_2, O_2, s_2') \in \mathcal{T}_2 \\ &\wedge\; (\ell_1 \wedge \ell_2)^+ \cap \Gamma \subseteq (O_1 \cup O_2) \;\wedge\; (\ell_1 \wedge \ell_2)^- \cap \Gamma \cap (O_1 \cup O_2) = \emptyset \\ &\qquad \iff (s_1 s_2, \exists \Gamma \,.\, (\ell_1 \wedge \ell_2), (O_1 \cup O_2) \setminus \Gamma, s_1' s_2') \in \mathcal{T}' \; .\end{aligned}$$

Because ℓ_2 does not range over Γ, $\ell_2^+ \cup \Gamma = \emptyset$, $\ell_2^- \cup \Gamma = \emptyset$, and $\exists \Gamma \,.\, \ell_2 = \ell_2$. Furthermore, because of $\Gamma \cap O_2 = \emptyset$, we also have $(O_1 \cup O_2) \setminus \Gamma = O_1 \setminus \Gamma \cup O_2$, $\Gamma \cap (O_1 \cup O_2) = \Gamma \cap O_1$, and $\ell_1^+ \cap \Gamma \subseteq (O_1 \cup O_2) \iff \ell_1^+ \cap \Gamma \subseteq O_1$. Thus, $\mathcal{T}' = \mathcal{T}''$. □

7.5.1 Proof for Theorem 5

For readability of the proof, we first consider only aspects without advice programs, and show that the theorem also holds for aspects with advice programs afterwards.

Let $\mathcal{A}_R = \{A_{R1} \dots A_{Rm}\} \subseteq \mathcal{A}$ be the recovery aspects in $\mathcal{A}$, with $A_i \in \mathcal{A}_R = (P_{JP_i}, (\mathit{recovery}, O_{adv_i}, P_{rec_i}))$ and $\mathcal{R}_i = \{r_{i,1} \dots r_{i,n_i}\}$ A_i's recovery states.

Because the parallel product is commutative we have

$$\begin{aligned}&\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_i} \| P_{JP_{i+1}} \| \dots \| P_{JP_n} \| P_{rec_{R1}} \| \dots \| P_{rec_{Rm}}) \\ &\qquad = \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_{i+1}} \| P_{JP_i} \| \dots \| P_{JP_n} \| P_{rec_{R1}} \| \dots \| P_{rec_{Rm}}) \; .\end{aligned}$$

We note

$$\begin{aligned}&P_{i+2} = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T}) = \\ &\qquad \boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n} \| P_{rec_{R1}} \| \dots \| P_{rec_{Rm}}) \triangleleft^{?}_{JP_n} adv_n \dots \triangleleft^{?}_{JP_{i+2}} adv_{i+2} \; .\end{aligned}$$

P_{i+2} is the program in that we weave A_i and A_{i+1} in different orders. We distinguish three cases: either (1) A_i and A_{i+1} are both *toInit* or *toCurrent* aspects, or (2) they are both *recovery* aspects, or (3) one is a either a *toInit* or *toCurrent* aspect, and the other a *recovery* aspect.

Case (1). Both A_i and A_{i+1} are either *toInit* or *toCurrent* aspects. Then $P_{i+2} \triangleleft_{JP_{i+1}} adv_{i+1}$ yields an automaton $P_{i+1} = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$(s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O \implies (s, \ell, O, s') \in \mathcal{T}'$$
$$(s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \implies (s, \ell, O_{adv_{i+1}}, targ_{i+1}(s)) \in \mathcal{T}',$$

where $targ_i(s)$ refers to the *targ* function used by the weaving of adv_i as defined in Definition 18, i.e. the target state of the advice transitions of adv_i.

$P_{i+1} \triangleleft_{JP_i} adv_i$ yields an automaton $P_i = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T}'')$, where $\mathcal{T}''$ is defined as follows:

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \notin O\big) \implies (s, \ell, O, s') \in \mathcal{T}'' \tag{7.1}$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \notin O\big) \implies (s, \ell, O_{adv_{i+1}}, targ_{i+1}(s)) \in \mathcal{T}'' \tag{7.2}$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O \wedge JP_i \in O\big) \implies (s, \ell, O_{adv_i}, targ_i(s)) \in \mathcal{T}'' \tag{7.3}$$
$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies (s, \ell, O_{adv_{i+1}}, targ_{i+1}(s)) \in \mathcal{T}'' \tag{7.4}$$

When we calculate $P_{i+2} \triangleleft_{JP_i} adv_i \triangleleft_{JP_{i+1}} adv_{i+1}$, we obtain the same automaton, except for transitions (7.4), which are defined by

$$\big((s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \wedge JP_i \in O\big) \implies (s, \ell, O_{adv_i}, targ_i(s)) \in \mathcal{T}'\big) .$$

Transitions (7.4) are exactly the join point transitions that are in $jpTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n}), JP_i) \cap jpTrans(\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n}), JP_{i+1})$. By precondition, there were no such transitions in $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n})$. Because we require in Definition 15 that all the JP_j outputs occur nowhere else, JP_i and JP_{i+1} cannot be contained in a O_{adv_j}, thus no transition of type (7.4) has been added by the weaving of $\triangleleft_{JP_n} adv_n \dots \triangleleft_{JP_{i+2}} adv_{i+2}$.

Thus, $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n}) \triangleleft_{JP_n} adv_n \dots \triangleleft_{JP_{i+2}} adv_{i+2} \triangleleft_{JP_{i+1}} adv_{i+1} \triangleleft_{JP_i} adv_i$ and $\boldsymbol{\mathcal{P}}(P, P_{JP_1} \| \dots \| P_{JP_n}) \triangleleft_{JP_n} adv_n \dots \triangleleft_{JP_{i+2}} adv_{i+2} \triangleleft_{JP_i} adv_i \triangleleft_{JP_{i+1}} adv_{i+1}$ are syntactically equal. Weaving $\triangleleft_{JP_{i-1}} adv_{i-1} \dots \triangleleft_{JP_1} adv_1$ in both expression thus yields the same result.

Case (2). Both A_i and A_{i+1} are *recovery* aspects. We first show that

$$P_{i+2} \triangleleft^R_{JP_{i+1}, \mathcal{R}_{i+1}} adv_{i+1} \triangleleft^R_{JP_i, \mathcal{R}_i} adv_i \sim P_{i+2} \triangleleft^R_{JP_i, \mathcal{R}_i} adv_i \triangleleft^R_{JP_{i+1}, \mathcal{R}_{i+1}} adv_{i+1} .$$

$P_{i+2} \triangleleft^R_{JP_{i+1}, \mathcal{R}_{i+1}} adv_{i+1}$ yields an automaton $(\mathcal{Q}, s_0, \mathcal{I} \cup \mathcal{R}ec_{i+1}, \mathcal{O}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$(s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \implies (s, \ell \wedge rec_{i+1,k} \wedge SR(i+1, k), O_{adv_{i+1}}, r_{i+1,k}) \in \mathcal{T}' \tag{7.5}$$
$$(s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \in O \implies (s, \ell \wedge SR(i+1, 0), O \setminus \{JP_{i+1}\}, s') \in \mathcal{T}' \tag{7.6}$$
$$(s, \ell, O, s') \in \mathcal{T} \wedge JP_{i+1} \notin O \implies (s, \ell, O, s') \in \mathcal{T}', \tag{7.7}$$

where $SR(i,k) = \bigwedge_{j=k+1..n_i} \overline{rec_{i,j}}$.

$P_{i+2} \triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1} \triangleleft^R_{JP_i,\mathcal{R}_i} adv_i$ then yields an automaton $(\mathcal{Q}, s_0, \mathcal{I} \cup \mathcal{R}ec_i \cup \mathcal{R}ec_{i+1}, \mathcal{O}, \mathcal{T}'')$, where $\mathcal{T}''$ is defined as follows:

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \notin O\end{array} \implies (s, \ell \wedge rec_{i+1,k} \wedge SR(i+1,k), O_{adv_{i+1}}, r_{i+1,k}) \in \mathcal{T}'' \tag{7.8}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \notin O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge rec_{i,k} \wedge SR(i,k), O_{adv_i}, r_{i,k}) \in \mathcal{T}'' \tag{7.9}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge rec_{i+1,k} \wedge SR(i+1,k), O_{adv_{i+1}}, r_{i+1,k}) \in \mathcal{T}'' \tag{7.10}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \notin O\end{array} \implies (s, \ell \wedge SR(i+1,0), O \setminus \{JP_{i+1}\}, s') \in \mathcal{T}'' \tag{7.11}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \notin O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge SR(i,0), O \setminus \{JP_i\}, s') \in \mathcal{T}'' \tag{7.12}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge SR(i+1,0) \wedge SR(i,0), O \setminus \{JP_{i+1}, JP_i\}, s') \in \mathcal{T}'' \tag{7.13}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \notin O \wedge JP_i \notin O\end{array} \implies (s, \ell, O, s') \in \mathcal{T}'', \tag{7.14}$$

Transitions 7.8 to 7.10 consider advice transitions. When we weave the advice the other way round, i.e. when we calculate $P_{i+2} \triangleleft^R_{JP_i,\mathcal{R}_i} adv_i \triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1}$, Transitions 7.10 are different, namely

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge rec_{i,k} \wedge SR(i,k), O_{adv_i}, r_{i,k}) \in \mathcal{T}'' \ .$$

However, Transitions 7.10 are excluded by the precondition, because they emit both JP_{i+1} and JP_i. Transitions 7.11 to 7.13 are the join point transitions that are not modified because no *rec* signal is present, and their combination with unmodified transitions. The weaving order does not influence them, because they do not remove the outputs, and thus the following aspect still finds its join point signal and modifies the transition. Transitions 7.14 are the transitions that are not modified by either aspect.

Thus, we have

$$P_{i+2} \triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1} \triangleleft^R_{JP_i,\mathcal{R}_i} adv_i \sim P_{i+2} \triangleleft^R_{JP_i,\mathcal{R}_i} adv_i \triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1} \ .$$

We now have to show changing the order of weaving $\triangleleft^I$ and $\triangleleft^M$ has the same result. $\triangleleft^I$ only replaces *REC* with *in* signals, and is thus not influenced by the weaving order, because the *REC* signals are different for each aspect. The $\triangleleft^M$ operator first creates a memory automaton for each aspect. The memory automata $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$ only depend on the recovery states $\mathcal{R}$, which are determined by $\triangleleft^R$, and do thus not depend on the weaving order of $\triangleleft^M$.

Let $P_R = P_{i+2}\triangleleft^? adv_{i+1} \ldots \triangleleft_1^? adv_1 \triangleleft^I_{\mathcal{R}_{Rm}} \ldots \triangleleft^I_{\mathcal{R}_{R1}} \triangleleft^M_{\mathcal{R}_{Rm}} \ldots \triangleleft^M_{\mathcal{R}_{i+2}}$. We have

$$\begin{aligned}
&P_R\triangleleft^M_{\mathcal{R}_{i+1}}\triangleleft^M_{\mathcal{R}_i} \\
=&\big[\big((P_R\|\mathcal{M}_{i+1}) \setminus (\mathcal{R}ec_{i+1} \cup \mathcal{I}n_{i+1})\big)\|\mathcal{M}_i\big] \setminus (\mathcal{R}ec_i \cup \mathcal{I}n_i) && \text{by Definition 23} \\
\sim&\big[(P_R\|\mathcal{M}_{i+1}\|\mathcal{M}_i) \setminus (\mathcal{R}ec_{i+1} \cup \mathcal{I}n_{i+1})\big] \setminus (\mathcal{R}ec_i \cup \mathcal{I}n_i) && \text{by Lemma 2} \\
\sim&\big[(P_R\|\mathcal{M}_{i+1}\|\mathcal{M}_i) \setminus (\mathcal{R}ec_i \cup \mathcal{I}n_i)\big] \setminus (\mathcal{R}ec_{i+1} \cup \mathcal{I}n_{i+1}) && \text{trivially by Definition 11} \\
\sim&\big[\big((P_R\|\mathcal{M}_i) \setminus (\mathcal{R}ec_i \cup \mathcal{I}n_i)\big)\|\mathcal{M}_{i+1}\big] \setminus (\mathcal{R}ec_{i+1} \cup \mathcal{I}n_{i+1}) && \text{by Lemma 2} \\
=&P_R\triangleleft^M_{\mathcal{R}_i}\triangleleft^M_{\mathcal{R}_{i+1}}\ . && \text{by Definition 23}
\end{aligned}$$

We can apply Lemma 2 above because $(\mathcal{R}ec_{i+1} \cup \mathcal{I}n_{i+1}) \cap (\mathcal{R}ec_i \cup \mathcal{I}n_i) = \emptyset$. Weaving $\triangleleft^M_{\mathcal{R}_{i-1}} \ldots \triangleleft^M_{\mathcal{R}_{R1}}$ also preserves equivalence.

Case (3). Let A_i be either a *toInit* or a *toCurrent* aspect, and let A_{i+1} be a *recovery* aspect. $P_{i+2}\triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1}$ yields an automaton $(\mathcal{Q}, s_0, \mathcal{I} \cup \mathcal{R}ec_{i+1}, \mathcal{O}, \mathcal{T}')$, where $\mathcal{T}'$ is defined by Transitions 7.5 to 7.7.

$P_{i+2}\triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1}\triangleleft_{JP_i} adv_i$ yields an automaton $(\mathcal{Q}, s_0, \mathcal{I}\cup\mathcal{R}ec_{i+1}\cup\mathcal{R}ec_i, \mathcal{O}, \mathcal{T}'')$, where $\mathcal{T}''$ is defined by

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \notin O\end{array} \implies (s, \ell \wedge rec_{i+1,k} \wedge SR(i+1,k), O_{adv_{i+1}}, r_{i+1,k}) \in \mathcal{T}'' \tag{7.15}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \notin O \wedge JP_i \in O\end{array} \implies (s, \ell, O_{adv_i}, targ_i(s)) \in \mathcal{T}'' \tag{7.16}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge rec_{i+1,k} \wedge SR(i+1,k), O_{adv_{i+1}}, r_{i+1,k}) \in \mathcal{T}'' \tag{7.17}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \notin O\end{array} \implies (s, \ell \wedge SR(i+1,0), O \setminus \{JP_{i+1}\}, s') \in \mathcal{T}'' \tag{7.18}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \in O \wedge JP_i \in O\end{array} \implies (s, \ell \wedge SR(i+1,0), O_{adv_i}, s') \in \mathcal{T}'' \tag{7.19}$$

$$\begin{array}{r}(s,\ell,O,s') \in \mathcal{T} \\ \wedge JP_{i+1} \notin O \wedge JP_i \notin O\end{array} \implies (s, \ell, O, s') \in \mathcal{T}'', \tag{7.20}$$

If we calculate $P_{i+2}\triangleleft_{JP_i} adv_i\triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1}$, we obtain the same transitions, expect for Transitions 7.17, which become advice transitions of A_i, and Transitions 7.19, which disappear. Again, these are exactly the transitions that are forbidden by the precondition. Thus we have shown that

$$P_{i+2}\triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1}\triangleleft_{JP_i} adv_i \sim P_{i+2}\triangleleft_{JP_i} adv_i\triangleleft^R_{JP_{i+1},\mathcal{R}_{i+1}} adv_{i+1}\ .$$

As in Case (1), applying the rest of the advice to two equivalent programs yields two equivalent programs.

Advice Programs. The join point and recovery transitions are selected before weaving inserts advice programs. Thus, the transitions in the advice programs are not chosen as join point transitions, nor can any of the introduced states be a recovery state. Thus, the inserted advice programs are not modified by either $\triangleleft_{JP}$, $\triangleleft^{R}_{JP,\mathcal{R}}$ or $\triangleleft^{I}_{\mathcal{R}}$, and it thus does not matter in which order they are inserted.

Weaving an aspect with an advice programs modifies the existing transitions in exactly the same way as the aspect without the advice program. Furthermore, only join point transitions and recovery transitions are modified. Thus, adding advice programs does not influence aspect interference. □

7.5.2 Proof for Theorem 4

Theorem 4 is a consequence of Theorem 5. We show that

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

follows from

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

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

7.6 Related Work

Some authors discuss the advantages of sequential vs. joint weaving. Lopez-Herrejon and Batory [LHB05] propose to use sequential weaving for incremental software development. Colyer and Clement [CC04, Section 5.1] want to apply aspects to bytecode which already contains woven aspects. In AspectJ, this is impossible because the semantics would not be the same as weaving all aspects at the same time.

Sihman and Katz [SK03] propose SuperJ, a superimposition language which is implemented through a preprocessor for AspectJ. They propose to combine superimpositions into a new superimposition, either by sequentially applying one to the other, or by combining them without mutual influence, i.e. they propose the same kinds of weaving as Larissa. Superimpositions contain assume/guarantee contracts, which can be used to check if a combination is valid.

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

Sanen et al [STJ$^+$06] propose to classify aspect interactions in four categories ranging from “reinforcement” to “conflict”, and to document them in a standardized way.

Douence et al [DFS02, DFS04] present a mechanism to statically detect conflicts between aspects that are applied jointly. Their analysis detects all join points where two aspects want to insert advice. To reduce the detection of spurious conflicts, they extend their pointcuts with shared variables, and add constraints that an aspect can impose on a program. To resolve remaining conflicts, the programmer can then write powerful composition adaptors to define how the aspects react in presence of each other.

Pawlak et al [PDS05] present a way to formally validate precedence orderings between aspects that share join points. They introduce a small language, CompAr, in which the user expresses the effect of the advice that is important for aspect interaction, and properties that should be true after the execution of the advice. The CompAr compiler can then check that a given advice ordering does not invalidate a property of an advice.

Durr et al [DSBA05] propose an interaction analysis for Composition Filters. They detect when one aspect prevents the execution of another, and can check that a specified trace property is ensured by an aspect.

Balzarotti et al [BDCM05] use program slicing to check if different aspects modify the same code, which might indicate interference.

7.7 Conclusion

We present an analysis for aspect interference Larissa. First, we introduced an additional operator which *jointly* weaves several aspects together into a program. Instead of weaving a first aspect in a base program, and then a second in the result, as does the sequential weaving defined earlier, we determine first all the join point transitions in the program, and then apply the different pieces of advice one after the other. This kind of weaving leads to much less interference between aspects if both aspects were designed for the same base program. It is also closer to the way AspectJ weaves aspects, but with the difference that our joint weaving selects join points only in the base program, whereas AspectJ aspects select join points also in each other's advice. The definition of joint weaving was easy: because Larissa is defined modularly, we only had to rearrange the building steps of the weaving process.

We can analyze interference between jointly woven aspects with a simple parallel product of the pointcuts. When a potential conflict is detected, the user has to solve it by hand, if needed. In the examples we already studied, the conflicts were solved by simple modifications of the pointcuts.

It seems that the interference analysis for Larissa is quite precise, i.e. we can prove independence for most independent aspects. One reason for that are Larissa's powerful pointcuts, which describe join points statically, yet very precisely, on the level of transitions. Another reason is the exclusive nature of the advice. Two pieces of advice that share a join point transition never execute sequentially, but there is always one that is executed while the other is not. If the two pieces of advice are not equivalent, this leads to a conflict. Thus, as opposed to [DFS04], assuming that a shared join point leads to a conflict does not introduce spurious conflicts.

Chapter 8

Contracts for Aspects

8.1 Introduction

8.1.1 Synchronous Languages and Design-by-Contract

Design-by-Contract [Mey92] is a design principle, originally introduced for object-oriented systems, where a method is specified by a contract. A contract is a specification in form of an implication between an assumption clause and a guarantee clause. A method fulfills its contract if after its execution, the guarantee holds provided that the assumption was true when the program was called.

A contract describes obligations and benefits for both the caller and the programmer of the method with a contract. The caller of the method must ensure that the assumption holds, and obtains the guarantee in return. The programmer only needs to consider the cases where the assumption holds, but must ensure that the guarantee holds in these.

Contracts have been adapted to reactive systems by [MM04a, MM04b]. Reactive systems constantly receive inputs from their environment, and emit outputs to it. Therefore, it seems natural to let assumptions restrict the inputs, and let guarantees ensure properties on the outputs. Additionally, what a program is allowed to do often depends to a large extent on previous occurrences of signals. A convenient way to express such temporal properties over input and output traces are observers, which we introduced in Section 3.3. In this chapter, observers have a single output `err`, which is emitted to show that a trace is not accepted.

As an example for a contract of a reactive system, consider a mono-stable flip-flop (MFF), which has one input `a` and one output `b`, and emits two consecutive `b`s when it receives an `a`. We formalize this description with a contract. The guarantee consists of two automata, shown in Figures 8.1(b) and (c), which are composed in parallel. The automaton in Figure 8.1(b) guarantees that a single `b` is never emitted, and the automaton in Figure 8.1(c) guarantees that when `a` occurs while no `b` is emitted, `b` is emitted in the next instant. Figure 8.1(d) shows the product of Figure 8.1(b) and Figure 8.1(c), the guarantee of the contract.

We also introduce an assumption, that stipulates that `a`s always occur in pairs. It is shown in Figure 8.1(a). Finally, we introduce a sample implementation which fulfills the contract. It is shown in Figure 8.2.

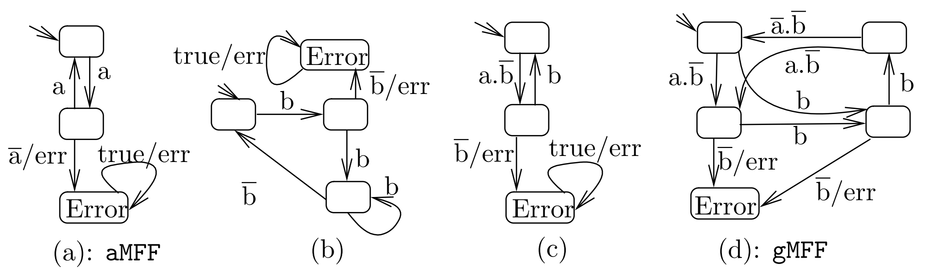


Figure 8.1: The contract for the MFF. The observers accept all traces that do not lead to state Error.

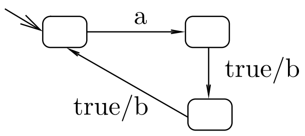


Figure 8.2: An implementation of the mono-stable flip-flop.

8.1.2 Combining Contracts and Aspects

AOP and design-by-contract cannot always be used concurrently. Obviously, the contract of a program may be invalidated when an aspect is applied to it. Consider the AspectJ example in Figure 8.3. The pointcut (line 7) intercepts calls to method `m` (line 4), and the around advice (lines 9–11) modifies the intercepted calls by adding `1` to the argument, then calling `m` through the proceed statement, and adding `1` to the result. After the aspect has been applied, neither the initial assumption (line 2) nor the initial guarantee (line 3) of `m` necessarily hold any longer. I.e., if `m(9)` is called, the proceed statement in the aspect will call `m` with argument `i=10`, which is outside the assumption, and we have thus no guarantee whatever about the return value. Even if `m` is called within its assumption, the proceed may always return 9, and the aspect will then return 10, which is outside the guarantee of `m`.

However, we can give a new contract for `m` in this case. To ensure that `m` is called according to its initial specification, the assumption must be changed to $i < 9$. On the other hand, the

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

Figure 8.3: Example of a contract in presence of an AspectJ aspect.

value returned by `m` may be higher than specified by the original guarantee in the presence of the aspect: we can only guarantee that $\backslash \mathit{result} < 11$. This, however, can only be guaranteed if `m` does not call itself recursively, or is otherwise affected by the aspect.

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

The remainder of the chapter is structured as follows: Section 8.2 defines contracts for Argos programs; Section 8.3 describes how to derive a new contract from a contract and an aspect, and prove the correctness of the new contract; Section 8.4 validates the approach on a larger example, modeling a tramway; Section 8.5 describes related work; and Section 8.6 concludes. A shorter version of this chapter has been published in [Sta07b], and a slightly different version of the tramway example in [SAM07] and [Sta07a].

8.2 Contracts for Argos

In Argos, contracts can conveniently be expressed with observers. The observers we use in contracts are slightly different from those used as pointcuts. Notably, once they start emitting their output err, they continue emitting it forever. Such an observer specifies a class of programs fulfilling a certain safety property, namely those programs where the observer never emits `err` when combined with them. Observers are formally defined as follows.

Definition 32 (Observer)**.** *An observer over $\mathcal{I} \cup \mathcal{O}$ is an automaton $(\mathcal{Q} \cup \{\mathit{Error}\}, q_0, \mathcal{I} \cup \mathcal{O}, \{$`err`$\}, T)$ which observes an automaton with inputs $\mathcal{I}$ and outputs $\mathcal{O}$. When an observer emits* `err`*, it will go to state Error, where it continuously emits* `err`*. A program P is said to* obey *an observer obs (noted $P \models obs$) iff $P \| obs \setminus \mathcal{O}$ produces no trace which emits* `err`*.*

Transitions leading to the Error state are called *error transitions.*

A contract specifies a class of programs with two observers, an assumption and a guarantee. We define it formally in Definition 33, where we use the trace combination defined in Definition 4. $\diamond$ denotes the trace for a single output `err` that always emits `false`, i.e. $\diamond(\texttt{err})[n] = \texttt{false}$ for all n.

Definition 33 (Contract)**.** *A contract over inputs $\mathcal{I}$ and outputs $\mathcal{O}$ is a tuple (A,G) of two observers over $\mathcal{I} \cup \mathcal{O}$, where A is the assumption and G is the guarantee. A program P fulfills a contract (A,G), written $P \models (A,G)$, iff for a pair of traces (it, ot) we have*

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

Definition 33 states that for a program P to fulfill a contract (A, G), all its traces that are accepted by A must also be accepted by G.

Intuitively, a guarantee G should only restrict the outputs of a program and an assumption A should only restrict the inputs. We do not require this formally, but contracts which do not respect this constraint are of little use. Indeed, if G restricts the inputs more than A, it follows from Definition 33 that there exists no program P s.t. $P \models (A,G)$. Conversely, a program is usually placed in an environment E, s.t. $E \models A$. If A restricts the outputs, no such E exists, as the outputs are controlled by P.

8.3 Weaving Aspects in Contracts

We want to apply an aspect *asp* not to a specific program, as we did until now, but to a class of programs, defined by a contract C. We then want to obtain a new class of programs, defined by a contract C', such that $P \models C \Rightarrow P \triangleleft asp \models C'$. To construct C', we simulate the effect that the aspect has on a program as far as possible on the assumption and the guarantee observers of C. However, an aspect cannot be applied directly to an observer, because the aspect has been written for a program with inputs $\mathcal{I}$ and outputs $\mathcal{O}$, whereas for the observer, $\mathcal{O}$ are also inputs.

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

Except for aspect weaving, all of these steps are different for the assumption and the guarantee, as far as the error transitions are concerned. This is because the assumption and the guarantee have different functions in a contract: the assumption states which part of the program is defined by the contract, and the guarantee gives properties that are always true for this part. Indeed, a contract (A,G) can be rewritten as $(\texttt{true}, A \Rightarrow G)$. Thus, the assumption can be considered as a negated guarantee.

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

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

8.3.1 Formal Definitions

This paragraph describes the weaving of aspects into contracts in detail, and illustrates it on our running example. First, Definition 34 defines the transformation of an observer into a NDA through two functions, one for guarantee observers and one for assumption observers.

Definition 34 (Observer to NDA Transformation)**.** *Let* $obs = (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T)$ *be an observer with an error state Error over inputs* $\mathcal{I}$ *and outputs* $\mathcal{O}$*, with* $\mathcal{I} \cap \mathcal{O} = \emptyset$*.* $ND_G(obs) = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_G})$ *defines a NDA, where* T_{ND_G} *is defined by* $(s, \ell_{\mathcal{I}} \wedge \ell_{\mathcal{O}}, \emptyset, s') \in T \Rightarrow (s, \ell_{\mathcal{I}}, \ell_{\mathcal{O}}^{+}, s') \in T_{ND_G}$*.* $ND_A(obs) = (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I}, \mathcal{O}, T_{ND_A})$ *defines a NDA, where* T_{ND_A} *is defined by* $(s, \ell_{\mathcal{I}} \wedge \ell_{\mathcal{O}}, o, s') \in T \Rightarrow (s, \ell_{\mathcal{I}}, \ell_{\mathcal{O}}^{+} \cup o, s') \in T_{ND_A}$*, where* $\ell_{\mathcal{I}} \in \mathcal{B}ool(\mathcal{I})$*,* $\ell_{\mathcal{O}} \in \mathcal{B}ool(\mathcal{O})$*, and* $\ell_{\mathcal{O}}^{+}$ *is defined as in Definition 11.*

Note that the transitions in *obs* which emit `err` (i.e. the error transitions) have no corresponding transitions in $ND_G(obs)$. In the guarantee, these transitions correspond to in-

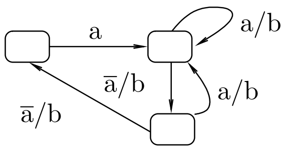


Figure 8.4: An implementation of the mono-stable flip-flop, with the retriggerable aspect `ret` applied to it.

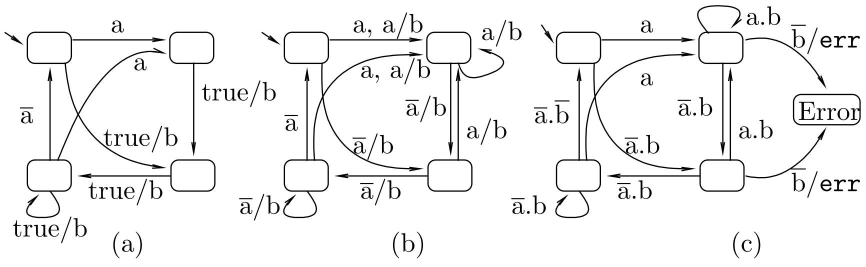


Figure 8.5: The weaving of the retriggerable aspect into the guarantee of the MFF. a: ND_G(gMFF), b: ND_G(gMFF)$\triangleleft$`ret`, c: OBS_G(ND_G(gMFF)$\triangleleft$`ret`).

put/output combinations which are never produced by the program and thus they must not be considered by the aspect.

As an example, consider the MFF introduced in Section 8.1.1. We now want to make the MFF re-triggerable, meaning that if the MFF receives an `a` that is emitted during several following instants, it continues emitting `b`. We do this by applying the aspect `ret`$=$ (P_{JP}, (*toInit*, $\{b\}$, (a))) to the MFF, where P_{JP} =({S},S,{a,b},{*JP*}, {(S,a∧b,*JP*,S)}) is a pointcut which selects all occurrences of `a`∧`b` as join points. Figure 8.4 shows the result of applying `ret` to the MFF from Figure 8.2.

Now, consider the guarantee of the MFF in Figure 8.1 (d). Its transformation into a NDA as defined in Definition 34 is shown in Figure 8.5 (a). Note that the Error state and the transitions leading to it have disappeared, and that`b` is now an output. Thus, the transition label `b` has transformed to `true/b`, and label `a.`$\bar{\texttt{b}}$ to `a`.

In the assumption, on the other hand, the error transition correspond to inputs from the environment to which the program may react arbitrarily. If the aspect replaces these transitions in the assumption, they are also replaced in the program, and can thus be accepted from the environment by the woven program. Thus, error transitions are not removed in $\mathit{ND}_A(\mathit{obs})$, so that the aspect weaving can modify them. The transformation of the assumption of the MFF in Figure 8.1 is shown in Figure 8.6 (a).

We can now apply an aspect to a NDA. However, the input trace that selects the target states of the advice transitions may lead to several states, because the NDA are non-deterministic. Thus, for each join point transition, several advice transitions must be created, one for each target state. We give a modified definition for *toInit*/*toCurrent* advice below. Because *recovery* advice involves no traces, the definition for deterministic programs in Section 4.3.2 can also be applied to NDAs.

Definition 35 (*toInit*/*toCurrent* Advice Weaving for NDA)**.** *Let* $A = (\mathcal{Q}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ *be an automaton and* $adv = (\mathit{type}, O_{adv}, \sigma, P_{adv})$ *a piece of advice, with type* $\in \{\mathit{toInit}, \mathit{toCurrent}\}$, $\sigma : [0, ..., \ell_\sigma] \longrightarrow [\mathcal{I} \longrightarrow \{\texttt{true}, \texttt{false}\}]$ *a finite input trace of length* $\ell_\sigma + 1$, *and* $P_{adv} =$

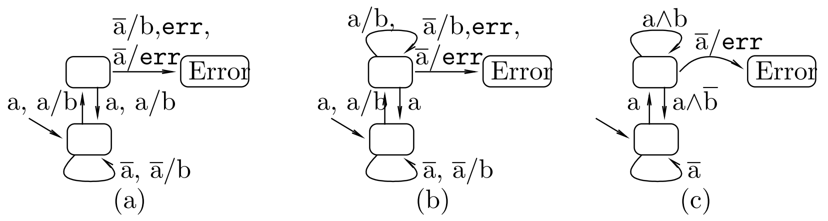


Figure 8.6: The weaving of the retriggerable aspect into the assumption of the MFF. a: ND_A(aMFF), b: ND_A(aMFF)◁`ret`, c: $OBS_A(ND_A$(aMFF)◁`ret`$)$.

$(\mathcal{Q}_{ins}, s_{0ins}, \mathcal{I}_{ins}, \mathcal{O}_{ins}, \mathcal{T}_{ins})$ an advice program with final state F.

Furthermore, let $TargSt = \{s | s = S\_step_A(s_0, \sigma, \ell_\sigma)\}$ if $type = toInit$ or $TargSt = \{s | \exists s' \in \mathcal{Q} \,.\, s = S\_step_A(s', \sigma, \ell_\sigma)\}$ if $type = toCurrent$ be the set of all target states, and let $InsSt = \{s_t | s \in \mathcal{Q}_{ins} \setminus \{F\}, t \in TargSt\}$ be the set of all states inserted by P_{adv}. Then, let the target state determination function $targ^{\mathrm{ND}}$ be defined as follows:

$$targ^{ND}(s) = \begin{cases} \{s' | s' = S\_step_A(s_0, \sigma, \ell_\sigma)\} & \text{iff } type = toInit \wedge s_{0ins} = F \\ \{s' | s' = S\_step_A(s, \sigma, \ell_\sigma)\} & \text{iff } type = toCurrent \wedge s_{0ins} = F \\ \{s' | s' = (s_{0ins})_{S\_step_A(s_0, \sigma, \ell_\sigma)}\} & \text{iff } type = toInit \wedge s_{0ins} \neq F \\ \{s' | s' = (s_{0ins})_{S\_step_A(s, \sigma, \ell_\sigma)}\} & \text{iff } type = toCurrent \wedge s_{0ins} \neq F \end{cases}$$

The advice weaving operator, $\triangleleft_{JP}$, weaves adv into A and returns the (possibly nondeterministic) automaton $A \triangleleft_{JP} adv = (\mathcal{Q} \cup InsSt, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T}')$, where $\mathcal{T}'$ is defined as follows:

$$(s, \ell, O, s') \in \mathcal{T} \wedge JP \notin O \implies (s, \ell, O, s') \in \mathcal{T}' \tag{8.1}$$

$$(s, \ell, O, s') \in \mathcal{T} \wedge JP \in O \wedge s'' \in targ^{ND}(s) \implies (s, \ell, O_{adv}, s'') \in \mathcal{T}' \tag{8.2}$$

$$(s, \ell, O, s') \in \mathcal{T}_{ins} \wedge s' \neq F \wedge t \in TargSt \implies (s_t, \ell, O, s'_t) \in \mathcal{T}' \tag{8.3}$$

$$(s, \ell, O, F) \in \mathcal{T}_{ins} \wedge t \in TargSt \implies (s_t, \ell, O, t) \in \mathcal{T}' \tag{8.4}$$

The $targ^{\mathrm{ND}}$ function for nondeterministic automata returns a set of the possible target states for the advice transitions. The advice transitions, defined by Equation (8.2), now point to each of these. The rest of the weaving is the same as in Definition 18. Note that there is no special treatment for error transitions: if an error transition is also a join point transition, it is replaced by an advice transition. The error transition thus disappears, as the outputs of the advice transition, O_{adv}, do not contain `err`.

Figure 8.5(b) and Figure 8.6(b) show the NDAs from our example with the retriggerable aspect woven into them. For both NDAs, the trace leads to a single state, thus only one advice transition is introduced per join point transition.

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

Definition 36 (NDA to guarantee transformation)**.** *Let $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, T)$ be a NDA. $OBS_G(nd) = (\mathcal{Q} \cup \{Error\}, q_0, \mathcal{I} \cup \mathcal{O}, \{$`err`$\}, T' \cup T'')$ defines an observer, where T' and T''*

are defined by

$$(s,\ell,o,s') \in T \Rightarrow (s, \ell \wedge \ell_o \wedge \ell_{\overline{\mathcal{O}\setminus o}}, \emptyset, s') \in T' \tag{8.5}$$

$$\begin{aligned}&(s,\ell,\emptyset,s') \notin T' \wedge s \in \mathcal{Q} \wedge \ell \text{ is a complete monomial over } \mathcal{I} \cup \mathcal{O}\\ &\qquad \Rightarrow (s,\ell,\{\texttt{err}\},\mathit{Error}) \in T''\end{aligned} \tag{8.6}$$

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

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

Definition 37 (NDA to assumption transformation)**.** *Let* $nd = (\mathcal{Q}, q_0, \mathcal{I}, \mathcal{O} \cup \{\texttt{err}\}, T)$ *be a NDA.* $OBS_A(nd) = (\mathcal{Q}, q_0, \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, T')$ *defines an observer, where* T' *is defined by*

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

Figure 8.5(c) and Figure 8.6(c) show the NDAs from our example transformed back into observers. As expected, the obtained guarantee in Figure 8.5(c) tells us that whenever the program receives an a, it emits b's the two following instants. The assumption, however, requires that if an a is emitted, it continues to be emitted until there is no b.

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

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

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

Definition 38 (Assumption Determinization)**.** *Let* M *be a NDA with outputs* $\{\texttt{err}\}$*.* $Det_A(M)$ *is a deterministic automaton such that*

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

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

Definition 39 (Guarantee Determinization)**.** *Let* M *be a NDA with outputs* $\{\texttt{err}\}$*.* $Det_G(M)$ *is a deterministic automaton such that*

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

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

Theorem 6. *Let P be a program and let (A, G) be a contract. Then,*

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

Theorem 6 first transforms the assumption and the guarantee into NDA with the respective operators, then applies the aspect to both and transforms the result back in observers, which are determinized. We prove it in the next section.

8.3.2 Proof of Theorem 6

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

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

$$\begin{aligned} A'{\triangleleft}asp &= OBS_A(ND_A(A){\triangleleft}asp), & A{\triangleleft}asp &= Det_A(A'{\triangleleft}asp), \\ G'{\triangleleft}asp &= OBS_G(ND_G(G){\triangleleft}asp), \text{ and} & G{\triangleleft}asp &= Det_G(G'{\triangleleft}asp) \;. \end{aligned}$$

***toInit/toCurrent* Advice.** We prove the theorem first for *toInit* and *toCurrent* advice, and therefore define the structure of some of these terms. Let

$$\begin{aligned} P &= (\mathcal{Q}_P, s_{P0}, \mathcal{I}, \mathcal{O}, \mathcal{T}_P), \\ asp &= (P_{JP}, (type, O_{adv}, \sigma, P_{adv})), \text{ with } type \in \{toInit, toCurrent\}, \\ P_{JP} &= (\mathcal{Q}_{P_{JP}}, s_{P_{JP}0}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T}_{P_{JP}}), \\ A &= (\mathcal{Q}_A \cup \{\text{Error}\}, s_{A0}, \mathcal{I} \cup \mathcal{O}, \{\mathtt{err}\}, \mathcal{T}_A), \\ G &= (\mathcal{Q}_G \cup \{\text{Error}\}, s_{G0}, \mathcal{I} \cup \mathcal{O}, \{\mathtt{err}\}, \mathcal{T}_G), \\ P{\triangleleft}asp &= (\mathcal{Q}_P \times \mathcal{Q}_{P_{JP}}, (s_{P0}, s_{P_{JP}0}), \mathcal{I}, \mathcal{O}, \mathcal{T}_{P\triangleleft}), \\ A'{\triangleleft}asp &= ((\mathcal{Q}_A \times \mathcal{Q}_{P_{JP}}) \cup \{\text{Error}\}, (s_{A0}, s_{P_{JP}0}), \mathcal{I} \cup \mathcal{O}, \{\mathtt{err}\}, \mathcal{T}_{A\triangleleft}), \text{ and} \\ G'{\triangleleft}asp &= ((\mathcal{Q}_G \times \mathcal{Q}_{P_{JP}}) \cup \{\text{Error}\}, (s_{G0}, s_{P_{JP}0}), \mathcal{I} \cup \mathcal{O}, \{\mathtt{err}\}, \mathcal{T}_{G\triangleleft}) \;. \end{aligned}$$

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

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

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

(p_n, pc_n), (a_n, pc_n) and (g_n, pc_n) are the states reached when executing (it, ot) for n steps on $P \lhd asp$, $A' \lhd asp$ and $G' \lhd asp$ respectively. The existential quantifiers before (a_n, pc_n) and (g_n, pc_n) are needed because $A' \lhd asp$ and $G' \lhd asp$ may be non-deterministic.

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

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

If $O\_step_{A \lhd asp}(it.ot, n) = \{\texttt{err}\}$, we are done. Otherwise, $O\_step_{A' \lhd asp}(it.ot, n) = \emptyset$ holds because of Definition 38, and we distinguish three cases separately, either no aspect applies, a *toInit* or a *toCurrent* aspect applies.

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

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

- Second case: $JP \in O\_step_{P_{JP}}(it.ot, n)$ and $type = toInit$.

  Let $p_\sigma = S\_step_P(s_{P0}, \sigma, l_\sigma)$ be the state in the P reached after executing σ, and let ς be a trace of length l_σ such that $\forall i \leq l_\sigma \ . \ \varsigma(i) = O\_step_P(s_{P0}, \sigma, i)$. Then, let $S\_step_{P_{JP}}(s_{P_{JP}0}, \sigma.\varsigma, l_\sigma) = \mathrm{pc}_\sigma$ be the state of the pointcut reached after executing σ. Then, we also have $S\_step_{P\lhd asp}((s_{P0}, s_{P_{JP}0}), it, n) = (p_\sigma, \mathrm{pc}_\sigma)$.

  All join point transitions in $G' \lhd asp$ (resp. $A' \lhd asp$) are replaced by transitions to all possible target states, thus there is a transition $t_{g'\lhd} \in \mathcal{T}_{G'\lhd}$ (resp. $t_{a'\lhd} \in \mathcal{T}_{A'\lhd}$) to a target state $(g_\sigma, \mathrm{pc}_\sigma)$ (resp. $(a_\sigma, \mathrm{pc}_\sigma)$) such that $S\_step_G(s_{G0}, \sigma.\varsigma, l_\sigma) = g_\sigma$ (resp. $S\_step_A(s_{A0}, \sigma.\varsigma, l_\sigma) = a_\sigma$). Because p_σ, a_σ and g_σ can be reached with the same trace (σ, ς) (resp. $(\sigma.\varsigma, \diamond)$ for a_σ and g_σ) from the initial state, $P(p_\sigma) \models (A(a_\sigma), G(g_\sigma))$ follows from $P \models (A, G)$.

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

- Third case: $JP \in O\_step_{P_{JP}}(it.ot, n)$ and $type = toCurrent$.

  Let $p_{n-1} = S\_step_P(s_{P0}, it, n-1)$, and let $p_\sigma = S\_step_P(p_{n-1}, \sigma, l_\sigma)$ be the state in the P reached after executing σ from the current state, and let ς be a trace of length l_σ such that $\forall i \leq l_\sigma \ . \ \varsigma(i) = O\_step_P(p_{n-1}, \sigma, i)$. Then, let $S\_step_{P_{JP}}(\mathrm{pc}_{n-1}, \sigma.\varsigma, l_\sigma) = \mathrm{pc}_\sigma$ be the state of the pointcut reached after executing σ. Then, we also have $S\_step_{P\lhd asp}((s_{P0}, s_{P_{JP}0}), it, n) = (p_\sigma, \mathrm{pc}_\sigma)$.

All join point transitions in $G'\triangleleft asp$ (resp. $A'\triangleleft asp$) are replaced by transitions to all possible target states, thus there is a transition $t_{g'\triangleleft} \in \mathcal{T}_{G'\triangleleft}$ (resp. $t_{a'\triangleleft} \in \mathcal{T}_{A'\triangleleft}$) to a target state $(g_\sigma, \mathrm{pc}_\sigma)$ (resp. $(a_\sigma, \mathrm{pc}_\sigma)$) such that $S\_step_G(g_{n-1}, \sigma.\varsigma, l_\sigma) = g_\sigma$ (resp. $S\_step_A(a_{n-1}, \sigma.\varsigma, l_\sigma) = a_\sigma$). Because p_σ, a_σ and g_σ can be reached with the same trace (σ, ς) (resp. $(\sigma.\varsigma, \diamond)$ for a_σ and g_σ) from p_{n-1},a_{n-1}, and g_{n-1}, $P(p_\sigma) \models (A(a_\sigma), G(g_\sigma))$ follows from $P(p_{n-1}) \models (A(a_{n-1}), G(g_{n-1}))$.

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

***recovery* Advice.** We now consider the weaving of *recovery* advice. First we redefine some of the definitions from the proof for *toInit* and *toCurrent* advice. Let

$$\begin{aligned}
asp &= (P_{JP}, adv),\\
adv &= (recovery, O_{adv}, P_{rec}, P_{adv})\\
P\triangleleft asp &= (\mathcal{Q}_P \times \mathcal{Q}_{P_{JP}} \times \mathcal{M}_P, (s_{P0}, s_{P_{JP}0}, s_{\mathcal{M}_P0}), In, \mathcal{O}, \mathcal{T}_{P\triangleleft}),\\
A'\triangleleft asp &= ((\mathcal{Q}_A \times \mathcal{Q}_{P_{JP}} \times \mathcal{M}_A) \cup \{\text{Error}\}, (s_{A0}, s_{P_{JP}0}, s_{\mathcal{M}_A0}), \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, \mathcal{T}_{A\triangleleft}), \text{ and}\\
G'\triangleleft asp &= ((\mathcal{Q}_G \times \mathcal{Q}_{P_{JP}} \times \mathcal{M}_G) \cup \{\text{Error}\}, (s_{G0}, s_{P_{JP}0}, s_{\mathcal{M}_G0}), \mathcal{I} \cup \mathcal{O}, \{\texttt{err}\}, \mathcal{T}_{G\triangleleft})\ .
\end{aligned}$$

We prove the same induction hypothesis as above, to which the same base case applies. We only reconsider the induction step, again from $n-1$ to n.

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

  We can apply the same argument as for non joint point transitions of *toInit*/*toCurrent* advice. The transitions t_p, t_a, and t_g are not affected by the product with the memory automaton and the encapsulation, because there condition does not depend on the *rec* signals.

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

  Let $m < n$ be the last instant in $1 \ldots n-1$ where REC is emitted, and let r_p, r_a, and r_g be the recovery states entered in m in $P\triangleleft^R adv$, $A\triangleleft^R adv$, and $G\triangleleft^R adv$ respectively. In m, $\mathcal{M}_P$, $\mathcal{M}_A$, and $\mathcal{M}_G$ enter their states r_p, r_a, and r_g respectively, and hence emit rec_{r_p}, rec_{r_g}, and rec_{r_a} in n. Thus, $P\triangleleft^R adv$, $A\triangleleft^R adv$, and $G\triangleleft^R adv$ take the advice transition leading to r_p, r_a, and r_g respectively. Because r_p, r_a, and r_g were all reached in the same instant m, we have $P(r_p) \models (A(r_a), G(r_g))$ by the induction hypothesis, and $r_g \neq$ Error also by the induction hypothesis.

If the aspect contains an advice program, it is inserted in P, A, and G in the same manner. Each time an advice transition is taken, the advice program is executed in all three programs, and they finish executing it at the same time, and return to the target state specified by σ. Thus, advice programs do not modify the induction hypothesis.

It follows from the induction hypothesis that

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

Controller Inputs:

inStation	the tram is in station
leaving	the tram wants to leave station
doorOpen	the door is open
doorClosed	the door is closed
askForDoor	a passenger wants to leave the tram
timer	the timer set by setTimer has run out

Controller Outputs:

doorOK	door is closed and ready to leave
openDoor	opens the door
closeDoor	closes the door
beep	emits a warning sound
setTimer	starts a timer

Gangway Inputs:

gwOut	the gangway is fully extended
gwIn	the gangway is fully retracted
askForGW	a passenger wants to use the gangway

Gangway Outputs:

extendGW	extends the gangway
retractGW	retracts the gangway

Helper Signals Outputs:

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

Figure 8.7: The interfaces of the controller and the gangway, and the helper signals.

and we have $(it.ot, \diamond) \in \mathit{Traces}(G' \triangleleft asp) \Rightarrow (it.ot, \diamond) \in \mathit{Traces}(G \triangleleft asp)$ by Definition 39. Thus, the theorem follows from the induction hypothesis. □

8.4 Example: The Tramway Door Controller

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

We implement the controller as an Argos program. We first develop a controller for a door without the gangway, and then add the gangway part with aspects. Figure 8.7 gives the in- and outputs of the controller with their specifications, and also the in- and outputs which are added by the gangway. The controller uses additional inputs, called Helper Signals, which are also shown in Figure 8.7 and are calculated from the original inputs by the program in Figure 8.8.

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

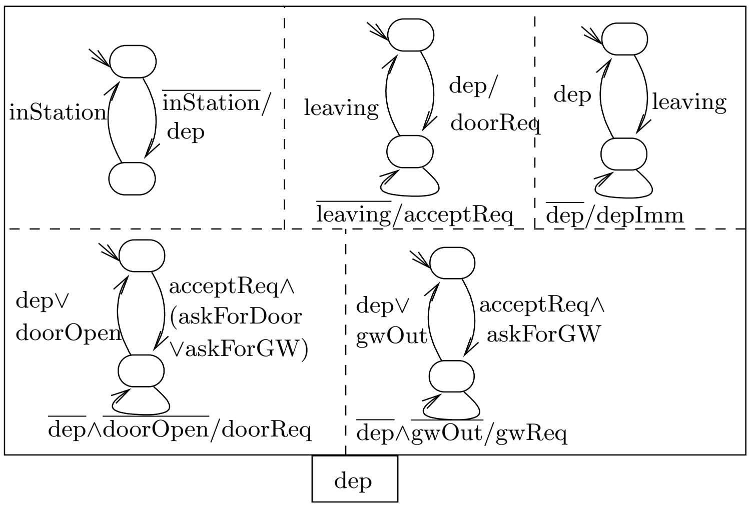


Figure 8.8: The automaton producing the helper signals.

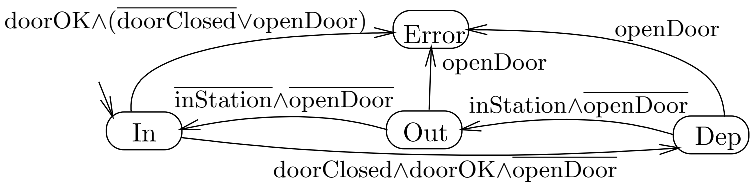


Figure 8.9: The guarantee of the contract of the controller.

guarantee of the contract is shown in Figure 8.9, it ensures that the controller emits `doorOK` only if the doors are closed, and `openDoor` only if the tram is in a station. The contract has also an assumption, which requires that the door behaves correctly (e.g., the door only opens if `openDoor` has been emitted). It is given in 8.10. An implementation of the controller, which fulfills the contract, is given in 8.11.

To formally verify that a tram door is always closed outside a station, we develop a model that describes the possible behavior of the physical environment of the controller, i.e. the door and the tramway. These models are expressed as Argos observers. The models for the tramway and the door are shown in Figure 8.12 and Figure 8.10 respectively. We then prove that the controller satisfies the contract, and that the contract in the environment never violates the safety property.

8.4.1 Adding the Gangway

Some doors include a gangway, which must also be controlled by the door controller. Two aspects are used to add support for the gangway to the controller: the extension aspect, that extends the gangway before the door is opened if a passenger has asked for it, and the retraction aspect, that retracts the gangway if it is extended and if the tram is about to leave.

The pointcut $P_{JP_{\text{ext}}}$ of the extension aspect selects all transitions where $openDoor \wedge doorReq \wedge doorClosed \wedge \overline{gwOut}$ is true, and the pointcut $P_{JP_{\text{ret}}}$ of the retraction aspect selects all transitions where $doorOK \wedge \overline{gwIn}$ is true.

Both aspects insert an automaton and return then to the initial state of the join point

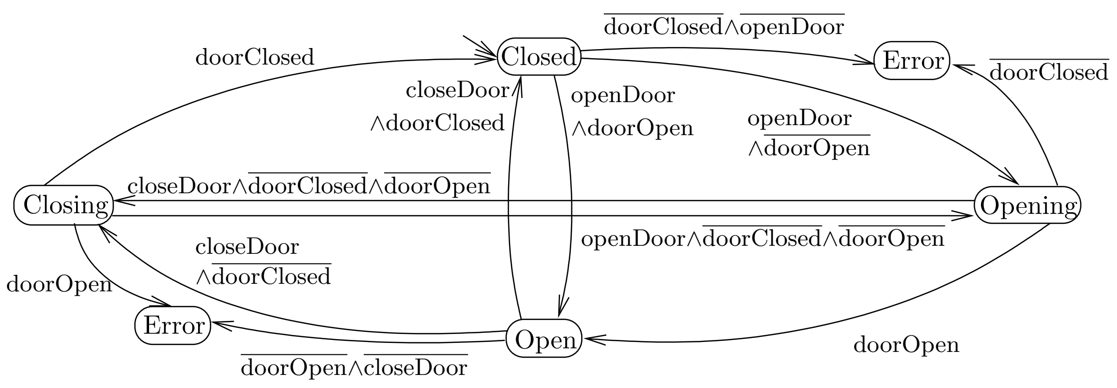

$\overline{\text{doorClosed}}\wedge\overline{\text{openDoor}}$
doorClosed
Closed
Error
$\overline{\text{doorClosed}}$
closeDoor
$\wedge$doorClosed
openDoor
$\wedge$doorOpen
openDoor
$\wedge\overline{\text{doorOpen}}$
Closing
closeDoor$\wedge\overline{\text{doorClosed}}\wedge\overline{\text{doorOpen}}$
Opening
openDoor$\wedge\overline{\text{doorClosed}}\wedge\overline{\text{doorOpen}}$
closeDoor
$\wedge\overline{\text{doorClosed}}$
doorOpen
Error
Open
doorOpen
$\overline{\text{doorOpen}}\wedge\overline{\text{closeDoor}}$

Figure 8.10: The model of the door, and also the assumption of the contract.

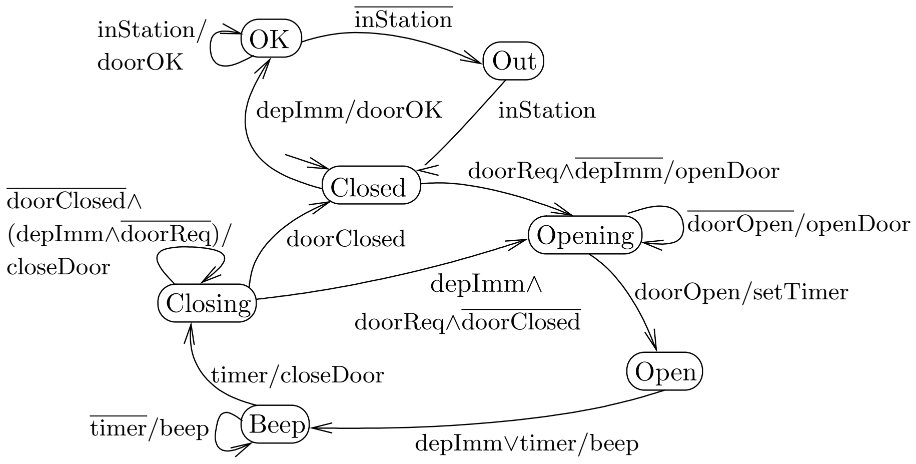

inStation/
doorOK
OK
$\overline{\text{inStation}}$
Out
depImm/doorOK
inStation
Closed
doorReq$\wedge\overline{\text{depImm}}$/openDoor
$\overline{\text{doorClosed}}\wedge$
(depImm$\wedge\overline{\text{doorReq}}$)/
closeDoor
doorClosed
Opening
$\overline{\text{doorOpen}}$/openDoor
Closing
depImm$\wedge$
doorReq$\wedge\overline{\text{doorClosed}}$
doorOpen/setTimer
Open
timer/closeDoor
$\overline{\text{timer}}$/beep
Beep
depImm$\vee$timer/beep

Figure 8.11: A sample controller for the tramway door.

transitions. The advice programs for the aspects are shown in Figure 8.13. The extension aspect is specified by $(P_{JP\text{ext}}, (\mathit{toCurrent}, , \emptyset, \epsilon, I_{\text{ext}}))$, and the retraction aspect by $(P_{JP\text{ret}}, (\mathit{toCurrent}, \{\mathit{retractGW}\}, \epsilon, I_{\text{ret}}))$, where ϵ denotes the trace of length zero.

8.4.2 Verification of the Woven Controller

We want to check that the new controller still verifies the safety property from above, i.e. that the doors are never open outside a station. Furthermore, we also want to verify two new safety properties related to the gangway: the first requires that the gangway is always fully retracted while the tram is out of station, and the second requires that the gangway is never moved when the door is not closed, to avoid possible injuries of the ankles of the passengers.

Therefore, we weave the aspects into the contract, using the method presented in Section 8.3. Thus, we obtain a new contract that holds for any controller which fulfills the contract, with the aspects applied to it. We also add a model of the gangway to the environment, which is the same as the model of the door, but where the door related signals have been replaced by their gangway related counterparts. Finally, we check then that the environment satisfies the new assumption, and that the new guarantee satisfies the safety requirements in the environment.

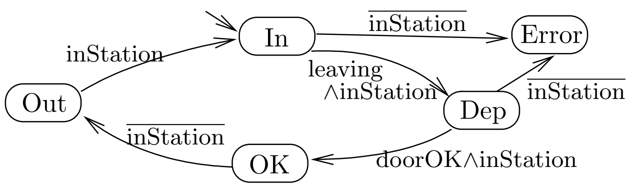

In
Error
Out
Dep
OK
inStation
$\overline{\text{inStation}}$
leaving
∧inStation
$\overline{\text{inStation}}$
$\overline{\text{inStation}}$
doorOK∧inStation

Figure 8.12: Model of the tramway.

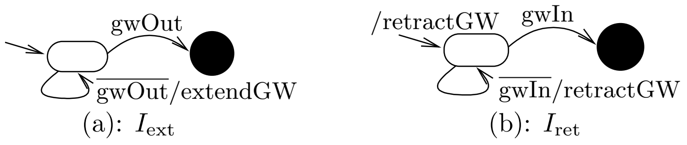

gwOut
$\overline{\text{gwOut}}$/extendGW
(a): I_{ext}
/retractGW
gwIn
$\overline{\text{gwIn}}$/retractGW
(b): I_{ret}

Figure 8.13: Advice programs for the extension (a) and the retraction (b) aspect.

An alternative to this modular approach is to verify directly that the sample controller with the aspects does not violate the given safety properties. One disadvantage of the non-modular approach is that the woven controller may be much bigger than the woven contract. To illustrate this problem, we verified the safety properties using our implementation (see Chapter 9, particularly Section 9.5.2 for contract checking). The source code of the door controller example is available at [Tra]. Verifying the woven program takes 11.0 seconds[1]. On the other hand, weaving the aspects into the guarantee of the controller contract and verifying against the environment takes 3.7 seconds[1], and verifying that the sample controller verifies the contract and verifying that the environment fulfills the assumption with the aspects takes < 0.5 seconds[1]. Thus, using this modular approach to verify the safety properties of the controller is significantly faster than verifying the complete program. Although the size of the woven controller is not prohibitive in this example, this indicates that larger programs can be verified using the modular approach.

8.5 Related Work

Goldman and Katz [GK07] modularly verify aspect-oriented programs using a LTL tableaux representation of programs and aspects. As opposed to ours, their system can verify AspectJ aspects, as tools like Bandera [CDH$^+$00] can extract suitable input models from Java programs. A main limitation of this approach is its restriction to *weakly invasive* aspects [Kat06], which only return to states already reachable in the base program, and which are only a subset of all AspectJ aspects. Larissa aspects, on the other hand, are all weakly invasive.

Clifton and Leavens [CL03a] noted before us that aspects invalidate the specification of modules, and propose that either an aspect should not modify a program's contract, or that modules should explicitly state which aspects may be applied to them.

Douence et al [DFS04] propose *applicability conditions* for aspects, that specify a class of programs to which an aspect can be applied, but do not give a property that is true for the woven program.

A number of authors (e.g. [Wam06, FBT06]) use aspects to implement design-by-contract tools for object-oriented programs, but do not consider aspects.

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

8.6 Conclusion

We proposed a way to show how a Larissa aspect modifies the contract of a component to which it is applied. This allows us to calculate the effect of an aspect on a specification instead of only on a concrete program. This approach has several advantages. First, aspects can be checked against contracts even if the final implementation is not yet available during development. Second, if the base program is changed, the woven program must not be re-verified, as long as the new base program still fulfills the contract. Third, woven programs can be verified modularly, which may allow to verify larger programs, as indicates the example in Section 8.4.

We believe that the approach is exact in that it gives no more possible behaviors for the woven program than necessary. I.e., for a contract C and a trace $t \in \mathit{Traces}(C \triangleleft \mathit{asp})$, there exists a program P such that $P \models C$ and $t \in \mathit{Traces}(P \triangleleft \mathit{asp})$. Because observers also accept traces that cannot be produced by finite state automata (such as `abaabaaab...`), we must in addition restrict the traces t to those that can be produced by a finite state automaton. This remains however to be proven.

Another interesting direction for future work would be to derive contracts the other way round. Given a contract C and an aspect asp, can we automatically derive a contract C' such that $C' \triangleleft \mathit{asp} \models C$? This seems difficult, as their are many possible solutions for C', which may or may not make use of the aspect to fulfill C after asp has been applied.

Finally, the proposed approach is restricted to Argos and Larissa with Boolean signals. It would be interesting to see if this approach can be extended to programs with internal integer variables, or with integer in- and outputs.

Extending the contract weaving to automata with internal integer variables but only Boolean in- and outputs seems possible. Transforming an observer with internal integer variables into a NDA can be done as defined in Definition 34 without changes to the variables. We must however adopt the weaving algorithm for NDA to integer variables.

If we allow integer in- and outputs, we have the problem that we cannot create a transition in the NDA for each integer the observer accepts, as there may be an infinite number of them. We thus must use an NDA with transitions that, instead of assigning each integer output an exact value, can assign them a value out of a set. Then, we can transform a condition as $a > 0$ into an assignment $a = \{x|x > 0\}$. We must also be able to split this transition. E.g., if the pointcut has a transition with condition $a < 5$ the product must then split the transition in the NDA with the above assignment into two transitions with the assignments $a = \{x|5 > x > 0\}$ and $a = \{x|5 > 0\}$.

However, introducing integers makes many things undecidable. Thus, we would need to make conservative approximations, and contract weaving would likely become less exact than the version presented in this chapter.

Chapter 9

Implementation

9.1 Introduction

This chapter introduces the compiler for Argos and Larissa, which has been developed for practical experimentation with the language and the tools introduced in the previous chapters. We describe the syntax of the textual Argos and Larissa that the compiler understands, and the functionalities it offers. The compiler does not implement all the functionalities presented previously (notably the interference analysis from Chapter 7 is not supported), but it offers a richer set of functions than those presented in some other areas.

The compiler reads programs in Argos and Larissa, checks if they are correct, and produces different kinds of output. It can either pretty print the input program, transform it into a single flat automaton by applying the definition of the operators, produce Lustre [Hal93] code, or Fast automata [Fas]. Through the translation into Lustre, Argos programs can be transformed into executable code, and they can profit from the development environment available for Lustre, which includes e.g. powerful verification tools (e.g. Lesar [HLR92]) or input generators for testing (e.g. [RWNH98, RRJ07]). The compiler can also be used to perform some naive verification on flat automata itself, which is however much less powerful than the tools available for Lustre.

Both flat or hierarchical programs can be translated into Lustre. The possibility to translate a hierarchic programs is important, because flattening large Argos programs is not always possible. Indeed, when calculating the product of parallel automata, the number of states is increasing very fast, a problem that is known as *state explosion.* However, not all hierarchical Lustre programs generated by the compiler are legal, because Lustre does not allow cyclic dependencies between parallel programs, whereas these are legal in Argos as long as they have a deterministic and complete semantics. In our experience, however, most Argos program do not contain such cycles. A more important limitation is that programs with aspects must also be flattened, because aspects cannot be expressed modularly in Lustre. This makes compiling impossible if an aspect is applied to a large program. We discuss a possible solution for this problem in Section 9.6.

This chapter is structured as follows: Section 9.2 contains some notes on how the compiler is implemented, Section 9.3 presents the textual syntax of Argos, Section 9.4 presents the textual syntax of Larissa, and Section 9.5 presents different features, including support for integer variables and contract checking. Finally, Section 9.6 discusses a way to weave an aspect into a program without flattening it first. The compiler is available at [Lar].

9.2 Implementation of the Compiler

The compiler was originally developed by Jerôme Couston, Benjamin Dufour, and Karine Altisen as a compiler for Argos in 2003. Since then, it has been maintained and extended by myself, in a continued effort since then until the completion of this thesis. It is written in Java and AspectJ. It uses JavaCC [JCC] to create a parser for Argos and Larissa programs, and the JavaBDD library [JBD] for Boolean calculus. Although it is designed as a proof-of-concept implementation, it can handle programs of considerable size. The main limitation is the state space explosion that occurs when performing the parallel product of large automata. The compiler is limited to programs of about 100,000 states and 1,000,000 transitions, much less than what model checking tools can handle. However, to perform verification, hierarchical Argos programs can be translated into Lustre, which is connected to the model checker Lesar [HLR92].

9.3 Syntax of Argos

We start with introducing a simple example of a flat automaton, and then introduce the Argos operators.

9.3.1 A Simple Example

Figure 9.1 shows the code for a simple automaton realizing a modulo-two counter, that emits a `b` every two `a`.

```
main modulo2counter(a)(b)

process modulo2counter(a)(b){
  controller{
    init A
    A{}
    -> B with a;

    B{}
    -> A with a/b;
  }
}
```

Figure 9.1: The textual syntax for a simple 2-state automaton.

Argos programs consist of a collection of processes, which represent automata. In our example we have only one process called `modulo2counter`. It starts with the keyword `process`, followed by its name, `modulo2counter`, a list of its inputs, here only `a` and a list of its outputs, here b. The input and output variables scope over the body of the process, which follows between braces.

Here, the body contains a flat automaton. This is specified by the keyword `controller`. Between the following braces, we first specify the initial state (`init A`), then follows a list of states with their transitions. Each state of the automaton is specified by its name (here `A` and `B`). It can be refined between the following braces. After the braces follows a list of transitions for each state. Each transition starts with `->` and ends with a semicolon. Between them are the state to which the transition goes, the keyword `with` and the activation condition. If the transition has outputs, they follow after a slash after the condition. The activation condition is an expression ranging over the inputs and the constants `true` and `false`. Connectors are the logical "and", written as `&`, the logical "or", written as `|`, and the logical "not", written as a prefixed `~`. Parentheses can also be used to write an expression. The line `main modulo2counter(a)(b)` at the beginning of the file specifies which process is the top level automaton of the program, and names its parameters.

9.3.2 Parallel Product and Encapsulation

The automaton in Figure 9.2 uses the parallel product and the encapsulation. In the body of the process `instantaneousDialogue` two automata are put into parallel and the communication signal `b` is encapsulated. The outermost element, the encapsulation of `b`, is expressed by `internal b`. `b` can be used as an input or an output between the following braces. To express the parallel composition, we put the parallel operator `||` between two automata.

```
process instantaneousDialog(a)(o){
  internal b{
    controller{
      init A
      A{}
      -> B with a&c/b;
      B{}
    }
    ||
    controller{
      init C
      C{}
      -> D with b/o;
      D{}
    }
  }
}
```

Figure 9.2: Two automata combined in parallel, communicating with an encapsulated signal.

The compiler supports a feature that is not included in the formal definition of Argos, but that is useful in practice. The encapsulation can be modified by putting the keyword `exported` after `internal`. Then, the encapsulated signals will stay output signals after the encapsulation has been applied. In the above example, declaring `b` exported will mean that it is emitted together with `o`. It must then also be declared an output.

9.3.3 Process Calls

Processes can call other processes. The process in Figure 9.3 instantiates the one-bit counter from Figure 9.1 three times to obtain a three-bit counter. A process with the name `modulo2counter` and the corresponding interface must be declared in the same file, or in one which is imported.

```
process modulo8counter(a)(o){
  internal b,c{
    modulo2counter(a)(b)
    ||
    modulo2counter(b)(c)
    ||
    modulo2counter(c)(o)
 }
```

Figure 9.3: An three-bit counter, built from three one-bit counters.

9.3.4 Refinement

Figure 9.4 shows an automaton with one refined state. The refining automaton is given between the braces following the name of the state.

```
main R(a,x)(o)

process R(a,x)(o){
  controller{
    init X
    X{
      modulo2counter(a)(o)
    }
    ->Y  with x;

    Y{}
    -> X with x;
  }
}
```

Figure 9.4: An automaton with a refined state.

9.3.5 Inhibition

Figure 9.5 shows an example for the inhibition. Note that the compiler has not a `whennot`-operator as the formal definition of Argos, but a `when`-operator. It is followed by a Boolean expression between parenthesis and an Argos expression between braces. The Argos expression is only executed when the Boolean expression is true.

```
process inhibit(a,i)(o){
  when (i){
    modulo2counter(a)(o)
  }
}
```

Figure 9.5: An example for the inhibition.

9.4 Syntax of Larissa

Larissa aspects are declared similar to processes, and applied to programs with aspect calls. Aspect Calls are unary operators similar to the encapsulation or the inhibition, which apply an aspect to a program.

9.4.1 Aspect Calls

Aspects are applied to a program by an aspect call, an operator which takes an Argos program and an aspect as argument. The aspect is then applied to the program. An aspect call can occur inside a process, and must follow an Argos expression. It starts with `<|` followed by the name of the aspect, a list of the aspect's inputs and a list of its outputs, each of these between parentheses. E.g., `exampleProcess(a)(b,o) <| exampleAspect(a,b)(o,p)` applies the aspect `exampleAspect` to the process `exampleProcess`. Note that the aspect can use outputs of the automaton to which it is applied as inputs.

9.4.2 toInit Aspects

Aspects are entities similar to processes. Both *toInit* and *toCurrent* aspect are known as toInit aspects to the compiler, distinguished only from *recovery* aspects. Figure 9.6 shows an example for the declaration of an `toInit` aspect.

```
toInit exampleAspect(a,b,c)(d) {
  joinpoint jp;
  pointcut PC(a,b)(jp);
  adviceOutputs d;
  trace init(a&~b&~c.a&b&c);
}
```

Figure 9.6: A toInit aspect.

A toInit aspect is composed of the following elements. `toInit` specifies that this is a toInit aspect, followed by the name of the aspect, `exampleAspect`. Then follows the inputs of the aspect and the outputs of the aspect. In the body of the aspect, we first specify the join point signal, which is emitted by the pointcut to signal that we are in a join point. It is preceded by the keyword `joinpoint`. Then follows the pointcut, which must be a process call emitting the join point signal, and which is preceded by the keyword `pointcut`. The keyword `adviceOutputs` is followed by a list of outputs, that are the outputs of the advice transitions. This line is optional, if it is omitted, the advice transitions have no outputs.

The line `trace init(a&~b&~c&.a&b&c);` gives the trace that specifies the final state of the advice transition. After the keyword `trace` follows either `init`, `this`, or `target`. `init` specifies that this is a *toInit* advice, and `this` specifies that this is a *toCurrent* advice. `target` refers to a kind of advice that is not described in the rest of the document, to keep formalization easier to understand and because it is not used in any of the examples. It is similar to *toCurrent* advice, but the trace is executed from the target state the join point transition, instead of its source state. The actual trace follows between parentheses. The variables of one element of the trace are separated by an `&`, and the elements are separated by a dot. The trace in the example spans over the variables `a`, `b` and `c` and has a length of two.

9.4.3 Recovery Aspects

Figure 9.7 shows an example for the declaration of a recovery aspect. A recovery aspect

```
recovery restartProd(a,b,c)(){
  joinpoint JP;
  pointcut PC(a,b)(JP);
  recSig REC;
  recAut RecoveryStates(b,c)(REC);
}
```

Figure 9.7: A recovery aspect.

is declared with the keyword `recovery`, followed by the name and the in- and outputs. In the body, `joinpoint`, `pointcut`, and `adviceOutputs` have the same meaning as for toInit aspects. Instead of a trace, the recovery aspect contains a second observer which determines the recovery states. First, we specify the recovery signal, with the keyword `recSig`. A process call to the recovery automaton is the given after the keyword `recAut`. It must emit the recovery signal.

9.4.4 Inserting Advice Programs

The compiler also supports the insertion of advice programs. The specification of the advice programs is however slightly different from the one presented in the rest of the document. To insert an advice program, the programmer must insert a line like `insert inserted(advice,a)(back);` in the declaration of the aspect, before the specification of the trace. `insert` is a keyword indicating that an automaton is inserted, and `inserted(advice,a)(back)` is the process call to the automaton that is inserted. `advice` and `back` are special signals that must not be defined in the interface of the aspect. The process `inserted` should be of a special form: it should have an initial state with a single transition waiting for the input `advice`. When `advice` becomes true, it should start executing the advice. Instead of going to a final state F when the execution of the advice is finished, it should go back to the initial state, and emit the output `back`, to reactivate the program.

The weaving will then introduce a waiting state between the source state of the join point transition and the target state. The advice transition then goes to the waiting state and emits `advice`, and there is a transition from the waiting state to the target state with condition `back`. The advice program is then put in parallel with the woven program and `advice` and `back` are encapsulated. Thus, when a join point of an aspect with an advice program is reached, the woven program goes to the state to which the transition labeled by `advice` points, executes the advice program until `back` is emitted, and then resumes the execution of the woven program at the target state specified by the aspect.

9.5 Extensions

9.5.1 Integer Variables

9.5.1.1 Syntax

The ArgosCompiler also supports integer variables in Argos programs, but weaving Larissa aspects in such programs is not defined. Applying an aspect to an Argos program with integers may lead to undefined results, especially as far as the execution of the trace is concerned.

In a process declaration, integer variables can be declared in three different ways:

- as input variables, declared after the Boolean inputs, and each variable preceded by the keyword `int`,
- as output variables, declared after the Boolean outputs, and each variable preceded by the keyword `int`, and
- as internal variables, declared either in at the beginning of the body of a process, or in automata before the initial state declaration, with the syntax `int varName = initialValue;`.

The `int` keyword must also precede integer variables in process calls. In the main process call, output variables are also assigned their initial values: they are followed by an `=` and an initial value. Output and internal integer variables can be assigned values by transitions, where assignments of the form `varName = intExpression` can be placed among the outputs. `intExpression` are either integer constants, integer variables, integer variables preceded by the keyword `pre` (referring to the value of the variable at the previous instant), or two intExpressions combined by one of the operators +, -, or *.

Integer variables can be used in the assignments of the conditions using the syntax `intExpression < intExpression` or `intExpression > intExpression`.

Figure 9.8 shows an example for a Argos program with integer variables. When in state `On`, it emits the value of its internal counter variable `value`. `value` is increased by input variable `a` while `value` is smaller then 10, otherwise is it decreased by `a`.

Note that determining if a state is complete and deterministic is undecidable with integer variables. The ArgosCompiler thus only accepts a subset of all deterministic and complete

```
main counter(b, int a)(int o = 0)

process counter(b, int a)(int o){
    controller{

      int value = 0;

      init Off

      Off{}
      -> On with b/ o = value;

      On{}
      -> Off with b / o = 0;
      -> On with  ~b & pre value < 10/
              o = value, value = pre value + a;
      -> On with  ~b & pre value > 9 /
              o = value, value = pre value - a;
    }
}
```

Figure 9.8: An example program with integer variables.

programs. E.g., if a transition has the condition a < b + 3, the program is only accepted if all other transitions of the state include ~(a < b + 3) in their condition. However, the ArgosCompiler correctly determines determinism and completeness for expressions of the form `variable (<|>) constant`, which are sufficient in most cases.

Parallel processes must never have common integer output variables. This means an integer variable is always assigned a value by exactly one process. Integer variables cannot be encapsulated, but integer output variables of a process can be read within the process. Thus, all integer output variables are also internal variables.

9.5.1.2 Semantics

We informally describe the semantics of integer variables in Argos. It is close to the Lustre semantics, so that the translation into Lustre is feasible. Input variables are assigned a value from the environment at each instant, and internal and output integer variables a value by the program as follows. All output and internal integer variables have an initial value, which is defined for output variables in the main process call, and for internal variables in their declaration. If a transition is taken which assigns a new value to a variable, it is instantly assigned this new value, otherwise it keeps the value from the previous instant. The value from the previous instant can be read with the `pre` keyword. In the first instant, `pre` returns the initial value of the variable.

9.5.2 Support for Contracts

The Argos compiler has some dedicated support for working with contracts expressed using synchronous observers, as presented in Chapter 8.

9.5.2.1 Error States

When working with observers, the compiler assumes that all error states are named ERROR. Then, in a product, the error states are combined, so that there is always only one error state. This can make products with observers considerably smaller.

9.5.2.2 Observer Transformations

To support weaving an aspect into a contract, the compiler can transform observers into non-deterministic automata, and vice-versa. The operator `nondeterministic` does the first, and the operator `observer` the second. Each must be followed by the list of signals that are to be converted from inputs to outputs or the other way round.

```
process property(i1,i2,o1,o2)(err){
  ...
}

process property-nda(i1,i2)(o1,o2){
  nondeterministic o1,o2{
    property(i1,i2,o1,o2)(err)
  }
}

process property-obs(i1,i2,o1,o2)(err){
  observer o1,o2{
    property-nda(i1,i2)(o1,o2)
  }
}
```

Figure 9.9: An example for the transformation of an observer into a non-deterministic automaton, and back.

Figure 9.9 shows an example. The observer `property` expresses some property over the signals `i1,i2,o1,o2`. In the process `property-nda`, it is transformed into a nondeterministic automaton with inputs `i1,i2` and outputs `o1,o2`. The process `property-obs` transforms it back into an observer, which is the same as the original `property`. The transformations are not the same for guarantee and assumption observers (see Definitions 34, 36, and 37). Both transformations are done as necessary for guarantees. To do the transformations for assumptions, the keyword `assumption` must be placed after `nondeterministic` and `observer`.

9.5.2.3 Contract Checking

Finally, the compiler can check a contract. Therefore, the following conventions must be followed. The assumption and the guarantee observer must be put in parallel to the program, and they must emit different error signals, e.g. `errA` and `errG`. Then, another automaton must be put in parallel which emits the output `fail` if `~errA&errG` is true. If the contract holds, `fail` is never emitted. If we now call the ArgosCompiler with the option `-contract`, it will check if a transition which emits `fail` is reachable in the flattened program, and if so, emit a trace leading to it.

9.5.3 File Inclusion

The ArgosCompiler disposes of a primitive mechanism to include process and aspect declarations from other files. Therefore, insert `include pathToFile1, ..., pathToFilen;` at the beginning of the an Argos file. The path are resolved from directory where the ArgosCompiler is executed. If the compiler cannot find an included file, it prints an error message and gives the current directory.

9.6 Towards Structure-Preserving Weaving

Problem Description. The most efficient way to compile large Argos programs into Lustre is to translate the hierarchic Argos program directly, without flattening it first. Indeed, flattening is impossible for large programs, because calculating the product of large parallel automata quickly leads to state explosion. However, we cannot translate aspects directly into Lustre, because they cannot be expressed there modularly. Furthermore, because aspect weaving is only defined on flat automata, we must flatten all programs to which an aspect is applied, which is impossible for larger programs.

This problem could be easily solved if aspects were distributive. Then, we could apply the aspects to the automata which form the leaves of an Argos expression, and thus avoid to calculate the parallel product that leads to state explosion. As an example, consider an aspect *asp*, two parallel automata A_1 and A_2, and some encapsulated signals Γ. If aspect weaving were distributive, $(A_1 \| A_2 \setminus \Gamma) \triangleleft asp = (A_1 \triangleleft asp \| A_2 \triangleleft asp) \setminus \Gamma$ would hold. This, however, is not possible, for two reasons:

1. The pointcut needs to know the outputs emitted by the *whole* program before it can decide whether the program is in a join point.
2. To execute the trace σ of a *toInit* or *toCurrent* aspect on one of the parallel automata A_1 or A_2, we need to know all the inputs of the automaton, including those in Γ, which are not defined in σ. The values of the signals in Γ for the execution of σ can only be defined by executing σ on both automata together.

Thus, we cannot achieve distributivity for Larissa aspects. However, full distributivity is not necessary to enable weaving for large programs, but it would be sufficient if we could avoid the calculation of the expensive parallel product. It would hence already be nice if the aspects could be woven into $A_1 \| A_2 \setminus \Gamma$ in such a way that we do not need to calculate the parallel product of A_1 and A_2. We present such a such a weaving algorithm in this section, and call it *structure-preserving weaving.*

Limitations of the Approach. The approach we present in this section has not been fully investigated. First and foremost, it is not implemented, and we thus have not tested if it really makes weaving larger programs possible. As we discuss below, the definitions suggest that in certain cases, we may pay for a reduced memory consumption with a prohibitive increase in compile time.

We think nonetheless that the approach is worthwhile, and that there are at least important special cases where it promises important gains, notably the cases where the second problem mentioned above does not apply, e.g. if there are no encapsulated signals or if the trace is empty.

The approach is also limited in other respects. We have not proven the equivalence of structure-preserving weaving with the weaving defined earlier. Furthermore, we do not define structure-preserving weaving for complete Argos and Larissa, but only for a subset. First, we only define the weaving of *toCurrent* aspects without advice programs, because this simplifies notation a great deal. Weaving *toCurrent* aspects is more complicated than weaving *toInit* or *recovery* aspects. Indeed, we believe that the other kinds of weaving do not pose any major difficulties that are not addressed in the weaving of *toCurrent* advice. The insertion of advice programs is independent of the preservation of the structure.

Second, we only consider Argos expressions that consist of parallel automata and some encapsulated signals. The parallel product and the encapsulation are the most important operators of Argos, and we believe that the approach can be extended to more complicated expressions and the other operators. This needs further investigation, however.

9.6.1 Structure-Preserving Weaving

Informal Explanation of the Approach. The two problems mentioned above are overcome by the use of additional communication signals, in the following way:

1. To determine if the program is in a join point, the poincut is executed in parallel with the program. The parallel components are modified to emit copies of the outputs they would have emitted without the aspect. The pointcut reads these copies, uses them to calculate if the program is in join point, and informs the parallel components about this. In turn, the components update their state and emit the outputs of the woven program depending on whether the program is in a join point or not.

2. For *toInit* aspects, the target state of the advice transitions can be determined statically. Therefore, the trace must be executed on the parallel automata during the weaving. For each element of σ, the values for the signals in Γ must be determined. If the encapsulation of Γ is deterministic and complete, there is exactly one solution, and the trace leads to a well-defined state in each automaton.
   For *toCurrent* aspects, the execution of σ in one of the parallel automata A_i depends on the values of Γ. These values cannot be determined once and for all during the weaving, but depend on the transitions taken by the automata in parallel with A_i, and thus on their current states. An advice transition may hence lead to different states, depending on the current states of the other parallel automata. Thus, the weaving must calculate all possible target states, and add advice transitions to them. Furthermore, the automata must communicate to each other their current states, such that each automaton knows which advice transition to take. There can then be a great number of

advice transition in each automaton, and calculating them is very expensive in certain cases.

Formal Definitions. In this section, we present a weaving algorithm that weaves an *toCurrent* aspect into some parallel automata that are also encapsulated.

Let $A_i = (\mathcal{Q}_i, s_{0i}, \mathcal{I}, \mathcal{O}, \mathcal{T}_i)$, $1 \leq i \leq n$ be automata, $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ a set of encapsulated signals, $asp = (P_{JP}, adv)$ an aspect with advice $adv = (toCurrent, O_{adv}, \sigma)$. We show how to weave the aspect into the expression $P = (A_1 \| \dots \| A_n) \setminus \Gamma$ without flattening it.

We first define the function *starget* which determines the target state of a trace from a set of initial states of each automaton. Because the parallel automata can interact through Γ, each step of the trace must be calculated together in all automata. *starget* is only defined if $(A_1 \| \dots \| A_n) \setminus \Gamma$ is deterministic and complete.

Definition 40 (*starget*)**.** *Let $P = (A_1 \| \dots \| A_n) \setminus \Gamma$ be an Argos expression, $A_i = (\mathcal{Q}_i, s_{0i}, \mathcal{I}, \mathcal{O}, \mathcal{T}_i)$, $1 \leq i \leq n$ be automata, $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ a set of encapsulated signals, and σ a trace over $\mathcal{I} \setminus \Gamma$. The target state function starget takes a trace σ and one state from each A_i, and returns another state from each A_i. It is defined as follows:*

$$starget_P(\sigma, s_1 \dots s_n) = \begin{cases} s_1 \dots s_n & \text{if } \sigma = \epsilon \\ starget_P(\sigma(1 \dots l_\sigma), s'_1 \dots s'_n) & \text{if } \forall i \leq n \;.\; \exists (s_i, \ell_i, O_i, s'_i) \in \mathcal{T}_i \;. \\ & \quad \sigma(0) \Rightarrow \ell_i \\ & \quad \wedge \ell_i^+ \cap \Gamma \subseteq \bigcup_{i \leq n} O_i \\ & \quad \wedge \ell_i^- \cap \Gamma \cap \bigcup_{i \leq n} O_i = \emptyset \\ \perp & \text{otherwise.} \end{cases}$$

starget is defined recursively over the σ. The first case is the base case: if the trace is empty, the start states $s_1 \dots s_n$ are returned.

The second case is the recursive case. Its condition selects transitions in $A_1 \dots A_n$ such that they fulfill the first element of the trace, $\sigma(0)$, and such that the condition of the encapsulation is fulfilled when these transitions are taken together. It has exactly one solution if P is deterministic and complete. The target states of the selected transitions $s'_1 \dots s'_n$ are then used to calculate the returned states with the rest of the trace $\sigma(1 \dots l_\sigma)$. If P is not deterministic, the third case applies and *starget* is not defined.

Advice Weaving. We now define how to weave the advice in the automata A_i. Each A_i must be prepared to execute an advice transition whenever it receives the signal *JP* from the pointcut. Furthermore, the advice transitions may lead to different target states, depending on the states in which are the parallel automata. To communicate these states, we add input signals $\bigcup_{j \leq n} \bigcup_{k \leq m_j} \{in_{jk}\}$ to A_i, where each automaton A_j has m_j states. Whenever an automaton A_j is in its state k, it emits signal in_{jk}. We also add these outputs to A_i. Furthermore, A_i emits a copy O' of the outputs O it would have emitted if the aspect were not activated. These signals are read by the pointcut.

Definition 41 (Structure-Preserving Advice Weaving)**.** *The structure-preserving weaving operator $\triangleleft^S$, weaves a piece of advice $adv = (toCurrent, O_{adv}, \sigma)$ into an automaton $A_i =$*

$(\mathcal{Q}_i, s_{0i}, \mathcal{I}, \mathcal{O}, \mathcal{T}_i)$, which is part of the expression $P = (A_1 \| \ldots \| A_n) \setminus \Gamma$, and returns an automaton $A_i \triangleleft^S adv = (\mathcal{Q}_i, s_{0i}, \mathcal{I} \cup \bigcup_{j \leq n} \bigcup_{k \leq m_j} \{in_{jk}\}, \mathcal{O} \cup \bigcup_{k \leq m_i} \{in_{ik}\}, \mathcal{T}_i')$, where $\mathcal{T}_i'$ is defined by

$$\begin{aligned}
&(s_{ik_i}, \ell, O, s_i') \in \mathcal{T}_i \implies \\
&\quad (s_{ik_i}, \ell \wedge JP \wedge \bigwedge_{j \leq n} in_{jk_j}, O_{adv} \cup O' \cup \{in_{ik_i}\}, starget_P(\sigma, s_{1k_1} \ldots, s_{nk_n})[i]) \in \mathcal{T}_i' && (9.1)\\
&\wedge \quad (s_{ik_i}, \ell \wedge \overline{JP}, O \cup O' \cup \{in_{ik_i}\}, s_i') \in \mathcal{T}_i' && (9.2)
\end{aligned}$$

where each automaton A_i has m_i states, and $O' = \{o' | o \in O\}$.

Transitions (9.1) are the advice transitions, they are taken when *JP* is true. There is one for every combination of the in_{jk} signals. *starget* is used to calculate the respective target states. Transitions (9.2) are not advice transitions, they are taken when *JP* is false. Note that both transitions emit a copy of the *original* outputs O, and the signal in_{ik} that tells the parallel automata in which state A_i is. Both O' and in_{ik} do not depend on whether the advice transition or the transition from the base program is taken.

Aspect weaving also affects the pointcut, but in a slightly different way than the parallel automata. Notably, the advice transitions must not emit O_{adv}, but must continue emitting *JP*. Thus, the weaving only changes the target state of the join point transitions, which depends on the outputs emitted by the base program when the σ is executed. These depend on the current state of the A_i, thus we add the in_{jk} signals to the inputs. To calculate it, we put P_{JP} in parallel with the A_i, and use *starget* on the resulting program. Furthermore, the pointcut no longer takes the original outputs of the base program $\mathcal{O}$ as inputs, but its copies $\mathcal{O}'$.

Definition 42 (Structure-Preserving Pointcut Weaving)**.** *The structure-preserving weaving operator for the pointcut, $\triangleleft^{S_{JP}}$, weaves a piece of advice $adv = (toCurrent, O_{adv}, \sigma)$ into a pointcut $P_{JP} = (\mathcal{Q}_{P_{JP}}, s_{0P_{JP}}, \mathcal{I} \cup \mathcal{O}, \{JP\}, \mathcal{T})$, which is applied to a parallel expression $(A_1 \| \ldots \| A_n) \setminus \Gamma$, and returns a pointcut $A_i \triangleleft^{S_{JP}} adv = (\mathcal{Q}_i, s_{0i}, \mathcal{I} \cup \mathcal{O}' \cup \bigcup_{j \leq n} \bigcup_{k \leq m_j} \{in_{jk}\}, \{JP\}, \mathcal{T}')$, where $\mathcal{T}'$ is defined by*

$$\begin{aligned}
(s, \ell, \emptyset, s') \in \mathcal{T} &\implies (s, \ell', \emptyset, s') \in \mathcal{T}' && (9.3)\\
(s, \ell, \{JP\}, s') \in \mathcal{T} &\implies (s, \ell' \wedge \bigwedge_{j \leq n} in_{jk_j}, \{JP\}, starget_P(\sigma, s, s_{1k_1} \ldots, s_{nk_n})[1]) \in \mathcal{T}' && (9.4)
\end{aligned}$$

where $P = (P_{JP} \| A_1 \| \ldots \| A_n) \setminus \Gamma$, each automaton A_i has m_i states, $\mathcal{O}' = \{o' | o \in \mathcal{O}\}$, and ℓ' is ℓ where each $o \in \mathcal{O}$ has been replaced by o'.

Transitions (9.3) are not advice transitions, and remain unchanged. Transitions (9.4) are advice transitions, their target state is changed. The new target state is calculated depending on the states of the A_i, with the *starget* function.

Aspect Weaving. Finally, we construct the complete program by putting the $A_i \triangleleft^S adv$ in parallel with $P_{JP} \triangleleft^{S_{JP}} adv$, and encapsulating the additional signals.

Definition 43 (Structure-Preserving Weaving). *Let $A_i = (\mathcal{Q}_i, s_{0i}, \mathcal{I}, \mathcal{O}, \mathcal{T}_i), 1 \leq i \leq n$ be automata, $\Gamma \subseteq \mathcal{I} \cup \mathcal{O}$ a set of encapsulated signals, and $asp = (P_{JP}, adv)$ an aspect, with $adv = (toCurrent, O_{adv}, \sigma)$. The structure-preserving weaving operator $\triangleleft^S$ is then defined by*

$$(A_1 \| \dots \| A_n) \setminus \Gamma \triangleleft^S asp = (A_1 \triangleleft^S adv \| \dots \| A_n \triangleleft^S adv \| P_{JP} \triangleleft^{S_{JP}} adv \\ \| dupl_{o_1} \dots dupl_{o_n}) \setminus \Gamma \cup O' \cup \{JP\} \cup \bigcup_{i \leq n} \bigcup_{k \leq m_i} \{in_{ik}\} \,.$$

Note that Definition 43 does not introduce any circular dependencies, which may lead to nondeterminism or incompleteness. The A_i only read each other's in_{ik} signals, which do not depend on any inputs. Furthermore, the $\mathcal{O}'$ and in_{ik} signals, which are emitted by the A_i automata and are read by the pointcut, do not depend on the JP signal, which is emitted by the pointcut.

9.6.2 Discussion

***toInit* and *recovery* Advice.** The other kinds of advice are much easier than *toCurrent* advice. When weaving *toInit* advice, we can determine the target states statically, by executing the *starget* function from Definition 40 from the initial states of the parallel automata. Thus, we can direct all advice transitions to the calculated target states, and there is no need to communicate the current states of the A_i, and thus no need to add the in_{jk} signals.

In the case of *recovery* aspects, we must first determine the recovery states of each of the A_i. This can be done the usual way, by performing a product with the recovery automaton P_{rec}. However, after performing the parallel product with P_{rec}, we cannot encapsulate the outputs, because these may be emitted by another A_i. Thus, P_{rec} may select more recovery transitions than actually exist. Next, we can build the memory automaton locally, in the usual way. Because we may have selected too many recovery transitions, it may be larger than necessary. However, this is much cheaper than calculating the parallel product of the A_i.

Complexity of the Weaving Algorithm. The weaving algorithm presented in this section is designed to make the weaving of aspects into large programs feasible, and it does indeed avoid the calculation of the parallel product and thus reduces memory consumption significantly. In certain cases, however, the weaving algorithm is expensive in time. We therefore discuss its complexity for the different kinds of aspects informally.

The most expensive part is the execution of the trace in the parallel automata. *toInit* aspects must calculate one trace per automaton using *starget*. *toCurrent* aspects, however, must calculate target states for all combinations of states of A_i, i.e. $\Pi_i|\mathcal{Q}_i|$ times, where $|\mathcal{Q}_i|$ denotes the number of states of A_i. This is the same complexity as calculating the product of the A_i, which also has $\Pi_i|\mathcal{Q}_i|$ states. However, in the structure-preserving weaving, this complexity is in *time*, and not in *space*.

Let us consider now the cost of the execution of *starget*. We must find the right combination of transitions in each step, such that the condition in the second case of Definition 40 is fulfilled. In the worst case, we must try all possible valuations for Γ, which is very expensive: in each A_i, there are up to $2^{|\Gamma|}$ possible transitions, where $|\Gamma|$ denotes the number of signals in Γ. To execute one step of *starget* on n parallel automata, we must hence check up to $2^{|\Gamma| \times n}$ combinations of transitions.

However, this worst case occurs only if there are circular dependencies between the parallel automata, in which case we cannot translate them into Lustre anyway. If there are no circular dependencies between the parallel automata, there is at least on automaton A_i that determines which subset of Γ it emits independently of the valuation of Γ, and then there is at least one automaton that determines which subset of Γ it emits depending only on A_i, and so on. Thus, there is a fixed order in which we can select the transitions in *starget*, and we do not need to try different combinations of transitions. Executing the trace on an parallel automaton has then the same cost as executing it on the flat program.

In the absence of circular dependencies, further optimizations are possible. Thus, *toCurrent* aspects also do not need to calculate traces for all possible combinations of states, and do not need all in_{ik} as communication signals. This, however, is not integrated in our weaving algorithm.

There are also some special cases in which structure-preserving weaving is very cheap. These are the absence of the encapsulation (i.e. $\Gamma = \emptyset$), and an aspect with an empty trace. The latter case is quite common, as shown by the examples in this document: the fast-cumulative aspect from Section 5.3.2 has an empty trace, as well as the Gateway aspects from the Tram example in Section 8.4.

Chapter 10

Related Work

Although aspect-oriented programming is a relatively young paradigm, a great amount of research has already been performed on this topic. We thus cannot presume to present all relevant work in this area, but we will try to discuss some work that is close to this thesis, and to compare it to our work. Aspects for automata languages are presented in Section 10.1, languages with stateful pointcuts in Section 10.2, formally-defined aspect languages and calculi in Section 10.3, and work on the reconciliation of aspect-oriented programming and modular reasoning in Section 10.4. Some more specific related work has already been presented in earlier chapters, namely work on using aspects to build product lines in Section 5.4, work on aspect interference in Section 7.6, and work on aspects and contracts in Section 8.5.

10.1 Aspects for Automata Languages

The aspect-oriented modeling community aims at representing aspects at higher level of abstraction. Many approaches (e.g. [CB05, SHU06, CvdBE07a]) propose to integrate representations of aspects into UML, including StateCharts. The Weavr approach [CvdBE07a, CvdBE07b] is probably closest to Larissa. It extends a tool for Model-Driven Engineering, where aspects are woven in models that can be translated automatically into code, as opposed to the other approaches, which model aspects that must be implemented manually. However, none of these approaches are formally defined, or have the semantic properties we are looking for. Due to the complexity of UML, giving such properties seems very difficult, if not impossible.

Sipma [Sip03] applies aspects to formally-defined automata languages. The author proposes a notion of aspect 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 and the simpler structure of programs, which are sets of parallel automata. Argos has a general notion of programs, with any level of operators.

10.2 Stateful Pointcut Models

A lot of authors have noticed before us that pointcuts which depend on the *history* of the execution of the program are very useful. In AspectJ, pointcuts can refer the execution history directly through the `cflow` construct, that can only depend on the stack. To write pointcuts in AspectJ that refer to the complete history of the execution, programmers must use the `if` construct. However, all the infrastructure to record the execution history must then be written manually in Java. This has many disadvantages: writing the code is cumbersome, the meaning of the pointcut is hidden, and formal analysis of the pointcut is impossible.

Therefore, many constructs have been proposed that allow pointcuts to explicitly refer to the execution history in a convenient form. These pointcuts collect state of their own, and can decide whether to match a join point depending on this state. They are hence often called stateful pointcuts.

These constructs differ in their expressive power, ranging from restricted regular expressions to Turing completeness. In general, reducing expressive power has the advantage that formal analysis of the aspects is easier, and the runtime overhead (in execution time or memory) is smaller. Furthermore, the different approaches have different ways of expressing the stateful pointcuts. As we will see below, many approaches use automata, others tail recursive languages or context-free grammars.

Many languages have the expressive power of regular languages, just as Larissa pointcuts. Larissa is special, however, because the base language has the same reduced expressive power. If we were to extend Larissa to integer variables, it would be natural to allow the pointcuts to have the same expressive power, too.

We now present different models of stateful pointcuts, ordered by their expressive power. Then, we present some program monitoring approaches, that resemble stateful pointcuts closely.

Stateful Pointcuts. In Arachne [DFL$^+$05], 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`). Due to this restriction, these aspects have an very small runtime overhead.

Vanderperren et al [VSCF05] extend the aspect language JasCo with stateful aspects with the power of regular expressions, but do not offer any formal analysis.

Douence et al [DFS02, DFS04, DFS05] propose Event-based AOP a simple aspect language with tail recursion to describe pointcuts with the power of regular languages. It is independent of the base language, as it only observes events in the execution of the base program, and inserts advice. The weaving and the composition of several aspects are a kind of synchronous product between base program and aspects. Because it has a restricted expressiveness, this language can be analyzed formally. Thus, [DFS04] discusses a powerful aspect interaction analysis.

In [DBNS06], the approach is extended concurrent EAOP, which models the concurrent execution of advice. The product between the base program and the aspects then only synchronize on a subset of the execution events. This is close to our approach because both model aspects for parallel programs. However, we are interested in a synchronous kind of parallelism, as it appears in synchronous languages, whereas CEAOP describes an asynchronous kind, that can model either the execution of asynchronous threads or of distributed programs.

Nguyen and Südholt [NS06] extend this approach to a more expressive pointcut language, using Visibly Pushdown Automata [AM04], which are more expressive than regular languages, but retain many of their formal properties. They illustrate that aspects can be analyzed for interaction, and that aspects can be proven to preserve important properties of base programs [NS07].

Walker and Viggers [WV04] propose a temporal extension of AspectJ with the power of context-free grammars, which is thus also situated between regular and Turing complete languages.

Ostermann et al [OMB05] propose a pointcut language based on logic languages, which can also refer the execution history. Klose and Ostermann [KO05] proposes to define pointcuts with particularly powerful predicates on traces, which may also refer to the future. This poses serious limitations: weaving is not guaranteed to terminate.

Property Enforcers. A number of approaches have been proposed for enforcing properties of programs. They do not label themselves as aspect languages, but are similar to Larissa and the approaches presented in this section in that an automaton runs in parallel of the program that can modify or abort the execution. The goal of the authors is more specific, however: they 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.

In [CF00], 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.

Schneider [Sch00] proposes the notion of *security automata*. 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 [LBW04, LBW05] 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.

10.3 Formal Semantics and Properties for AOP

Superimposition. Katz [Kat93] proposes superimpositions, a programming construct used to impose a global property on distributed asynchronous systems. Their motivation is quite similar to aspects, and they have the advantage of a formal definition. Indeed, Sihman and Katz [SK03] have proposed SuperJ, a language to implement superimpositions with aspects.

Superimpositions are classified into three categories, which qualify the effect of a superimposition on a program. *Spectative* superimpositions only observe program, but do not influence its execution, *regulative* superimpositions restrict the base program by forbidding certain executions, and all other superimpositions are *invasive*. These categories also apply to aspects, and Katz has extended them in [Kat06], where the invasive category has been split into *weakly* and *strongly invasive*. Those invasive superimpositions that always return control to the base program in a state that it could have reached on its own are weakly invasive.

Superimposition is placed in a setting similar to ours, although it considers *asynchronous* parallelism, whereas our parallel combination is synchronous. In the context of synchronous languages, a spectative aspect is merely the synchronous composition with an observer (see Section 3.3). Regulative aspects make no sense in our context, where programs are deterministic, complete, and cannot be terminated. Thus, there are no executions to forbid.

Larissa is a weakly invasive aspect language, because the return state is defined either by a trace that the base program could have executed, or is a state the program has already passed. Furthermore, all aspect languages that preserve a semantic equivalence of programs must be weakly invasive. Indeed, for a strongly invasive aspect languages, it is easy to give equivalent base programs that are distinguished by the aspect: let P be a module, and A an aspect that sets P into a state v, that P could not reach on its own. Let Q be the same module as P, but with `if state=`v` then X endif` inserted at some point in the code where the aspect returns control. P and Q are equivalent, because `state=`v is always `false`. However, once we apply A, `state=`v is true, and Q executes code `X`, whereas P does not.

AOP Core Calculi. A number of authors have presented different core calculi modeling aspect-oriented languages. Most are defined with operational small-step semantics, and extend functional or object calculi.

Dutchyn et al [DTK06] present an aspect-oriented extension to a higher-order language, where both pointcuts and advice are also higher-order.

Walker et al [WZL03, LWZ06] propose a core aspect language, that extends a typed lamdba calculus with labels and advice, which runs at certain labels. They also give an aspect language, minAML, that translate into the core language. Dantas et al [DWWW05, DW06] follow the same principle, but extend both the core calculus and the aspect language. [DWWW05] investigates the combination of aspect-oriented programming and polymorphism. [DW06] present harmless advice, which does not interfere with the main computation of base program, but can perform I/O or terminate the program. This kind of advice can perform many typical aspect tasks, and simplifies local reasoning in the base program.

Aldrich [Ald05] presents another simple function core calculus, TinyAspects, which just contains pointcuts and advice. This makes it easier to prove modularity properties on an extension, Open Modules (see Section 10.4).

Clifton and Leavens [CL06] propose MiniMAO$_1$, an aspect-oriented extension to an imperative, object-oriented core language. It models precisely the interaction between object-oriented and aspect-oriented programming, including the case where the aspect changes the target object of the advised call.

Jagadeesan et al [JJR03] propose an aspect calculus for a rich class-based language, which includes multi-threading, and proves the correctness of their weaving algorithm. Bruns et al propose μABC [BJJR04], a name based calculus, where aspects are the only primitive entity, and are used to simulate methods. They also give a translation of minAML [WZL03] and the language from [JJR03] into μABC.

Furthermore, Wand et al [WKD04] give a denotational semantics for an aspect oriented extension of an imperative language. Lämmel [Läm02] gives a big-step semantics for an extension to an object-oriented language. Andrews [And01] translates a simple imperative aspect language into process algebra.

Larissa 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 crosscut their parallel structure. It is quite different from the AspectJ-like before, after, and around advice that make sense mainly for sequential languages. [And01] is situated in framework similar to ours. However, we aim at defining aspects that *crosscut* the parallel structure of reactive programs, not at formalizing weaving for sequential programs by a kind of parallel composition.

10.4 AOP and Modular Reasoning

AspectJ does not preserve the encapsulation of Java classes. As we discussed and illustrated in Section 6.1, aspects can distinguish between implementations which are semantically equivalent, but implemented in a different manner. For example, calls to private methods of a class can be advised. Thus, a modification of a class as simple as inlining a private method or changing a private method's name may break the behavior of an aspect that has been applied to it. Programmers of base modules can thus not even perform such innocuous changes without having to verify that they do not affect an aspect. Thus, modular reasoning becomes impossible in the presence of aspects. This problem has been recognized by many authors, and different solutions have been proposed.

Clifton and Leavens [CL03b] propose to extend the concept of behavioral subtyping [DL95] to aspects. Thus, aspects must modify methods in such a way that the specification of the advised method still holds. Furthermore, they propose to split aspects into two categories, spectators and assistants. Spectators do not change the behavior of the advised methods. Assistants, on the other hand, may modify their behavior, and thus break the encapsulation of the module. They propose that base modules should be able to specify which assistants may be applied to them.

In Larissa, we follow a different approach: a module to which an aspect has been applied becomes a new module, with a new semantics. This is consistent with the design of Argos, where all operators transform the semantics of the module they are applied to, without introducing problems for local reasoning inside a module.

Aldrich [Ald05] proposes the use of *open modules*. Aspects can only advise external calls to methods in the interface of an open module, and *exported pointcuts*, which are part of the interface of the module. If the implementation of a module is changed, the exported pointcuts must be updated, such that they select the same join points as before. This is the responsibility of the programmer who maintains the module. For a small aspect calculus, Aldrich has shown that open modules preserve equivalence between programs. Onkingco et al [OAH^{+}06] have extended AspectJ with constructs that implement Open Modules.

This is similar to Larissa, where aspects can only advise the interface of modules. Larissa has, however, no special mechanism to add exported pointcuts to the interface of a module, while such a construct forms a central part of Open Modules. This can nonetheless be simulated in Larissa by adding an additional output to a module that corresponds to such a pointcut, and that is read by the pointcut of an aspect. However, our examples indicate that exporting additional pointcuts is not necessary, and that Larissa is expressive enough without them. This is an advantage, because base module programmers do not need to update the additional pointcuts when they change the program.

Sullivan, Griswold, et al propose *crosscutting interfaces* [GSS$^+$06, SGS$^+$05], a notion similar to open modules. They do not, however, propose the extension of an aspect language by a new module system, but rather design rules for aspects. The aspect programmer only uses pointcuts which are declared in the crosscutting interface of a base module. These pointcuts are declared and maintained by the base module programmer. This is similar to Open Modules, but aspects are not allowed to advise directly the public interface of the base program.

Aksit, Bergmans, et al propose *Composition Filters* [BA04], a kind of aspect-oriented programming that also respects the encapsulation of base components. Composition Filters intercept messages between objects, and can manipulate them in different ways. They thus refer only to the interface of the base component, but not to its implementation details.

In Argos, such filters can be constructed easily with the existing operators, by putting a filter program in parallel with the base program. The filter program than intercepts incoming and outgoing signals, similar to the encoding of advice in Argos illustrated in Section 3.4. This does not include the full expressiveness of Composition Filters, which e.g. can use wildcards.

Some authors [GB03, OMB05, RKA06] attack the problem from the opposite direction, by making pointcut languages more powerful, instead of restricting the power of aspects. They argue that this makes modular reasoning easier, because aspects with more expressive pointcuts can express precisely when they want to intervene, and refer less to intermediate structures that are likely to change. Their goal is less to preserve the encapsulation, but more to make aspects resistant to changes in the base program, including those changes that result in a different semantics of the base program, such that even equivalence preserving aspect languages cannot guarantee any formal properties.

Gybels and Brichau [GB03] propose a logic programming language as pointcut language. Ostermann et al [OMB05] also follow that line, and present Alpha, a powerful Prolog-like pointcut language, that can in addition refer a rich data model, including the syntax of the program, the heap, and the execution trace. Because these approaches can refer many implementation details, it is impossible to give formal properties regarding the preservation of the encapsulation. However, the examples of pointcuts that are presented by the authors as being more resistant to change often refer less to implementation details: e.g., "all fields that changed during the previous call to method m" is less implementation dependent than enumerating the private fields that are changed at the moment the aspect was written.

Larissa follows this approach insofar as it has a powerful pointcut language, but renounces all elements that would compromise the preservation of formal properties. We believe that this makes aspects also more independent of changes in the base program that modify its semantics, such that the preservation of the equivalence does not apply.

Kiczales and Menzini [KM05] argue that approaches that restricting the power of pointcuts constrain the obliviousness [FF00] of aspects too much. Instead, they propose *aspect-aware interfaces*. Theses are normal class interfaces, with information about advising aspects added to method signatures. Such interfaces do not constrain programmers, and are implemented in IDEs with support for aspects, such as AJDT [CCK03]. On the other hand, they do not protect the encapsulation of the base module, and they are only available once the whole system is assembled.

Chapter 11

Conclusion

11.1 Context

Aspect-oriented programming is a relatively new programming paradigm, which aims at encapsulating cross-cutting concerns. It is being applied with great success in many domains, and it is being researched very actively. One domain to which it has so far not been applied, however, is the domain of reactive systems. This thesis has taken a first step in this direction. In particular, we have investigated cross-cutting concerns and aspect-oriented programming in the context of synchronous languages, a family of domain-specific languages for reactive systems, which are widely used in industry. Because there was no prior work on AOP and synchronous languages, we used the simplest language of the family, Argos (presented in Chapter 3), for our experiments.

As we discussed in Section 2.2, reactive systems differ from usual computer systems in a number of ways, and this is reflected in their programming languages. First, such systems are in constant interaction with their environment, and their programming languages are specially adapted to an input/output style of programming. Second, reactive systems are also very often safety-critical, and need to be verified formally. Their programming languages are thus formally defined, and connected to verification tools. Furthermore, they often fulfill many tasks in parallel. Thus, programming languages usually offer an explicit means to combine program modules in parallel.

Synchronous languages (presented in Section 2.2.1) are specially designed to meet these requirements. They are formally defined, have a clear and simple semantics, and are restricted to programming constructs that make verification easy. They also have a well-defined and easy-to-use parallel composition operator, which is used as the main means for decomposition of programs. Argos also follows these lines. Its basic elements are Mealy automata, which can be composed with a number of operators, including the parallel composition.

The characteristics of reactive systems named above impose certain restrictions on the design of aspect-oriented languages for them, as we discussed in Section 2.3. First, they must be formally defined, and respect certain semantic properties, notably the encapsulation of base programs. Furthermore, aspects must crosscut the *parallel* structure of reactive programs, as opposed to most existing aspect languages, which apply to programs written in *sequential* languages.

11.2 Comments on the Contributions

11.2.1 Cross-cutting Concerns in Reactive Systems

A first contribution of this thesis consists in the identification of cross-cutting concerns in reactive systems. We found cross-cutting concerns in different types of reactive systems we modeled: a process controller in a juice factory (Sections 4.2.1 and 4.3.1), a door controller in a tram (Section 8.4), and the interface of a complex wristwatch (Chapter 5). The concerns we found are all functional, i.e. they express a part of the core functionality of the application. This is different from other aspect languages, where many popular concerns are non-functional.

One reason for the absence of non-functional concerns in our examples may be that we wanted semantic aspects, which in particular respect the encapsulation of the base program. This excludes many common non-functional aspects, such as logging, profiling, or the enforcement of programming conventions, which often need to look at the implementation of the base program. Another reason may be that there are less non-functional concerns in the typical reactive systems we used as examples, and which are representative for applications written in synchronous languages. Many typical non-functional cross-cutting concerns, such as security or transaction management, do not occur in such systems. Others are not relevant to programs written in synchronous languages, either because they are superfluous, such as synchronization, or deemed unsafe, such as exception handling.

11.2.2 Larissa, an Aspect Language for Argos

As a second contribution, presented in Chapter 4, we developed an aspect-oriented language for Argos, called Larissa, which is able to express the examples of cross-cutting concerns we identified. The examples show that Larissa encapsulates cross-cutting concerns that occur in reactive systems, and that it adds expressive power to Argos.

Furthermore, Larissa integrates well into Argos, and aspect weaving can be considered another Argos operator, because it fulfills all the necessary conditions: it preserves the equivalence between base programs, the determinism and completeness of base programs, and it is possible to combine aspect weaving freely with the other operators to build programs. The last condition is fulfilled by defining weaving in the same way as the other operators, as a translation of a flat automaton and an aspect into a flat automaton.

The previous conditions concern the preservation of important semantic properties, where the preservation of the equivalence is the most difficult to guarantee. It entails that the aspect can only refer to the interface of the base program, but not its implementation. The preservation of the equivalence is an interesting feature of Larissa, that most aspect languages do not offer.

Indeed, the fact that most aspect languages do not preserve the equivalence between programs has been recognized as a major drawback, and several solutions have been proposed (see Section 10.4). Those solutions that preserve the equivalence come at a price: usually the base module programmer must take responsibility for maintaining some pointcuts for the base module. Larissa does impose no such condition, and nevertheless the base programs stays strongly encapsulated.

A second interesting point of Larissa is that it extends a *parallel* programming language. The existence of an explicit parallel composition in the base language had important consequences for the design of Larissa, because many typical uses of aspects can be expressed modularly with the existing operators in Argos. Larissa aspects thus look quite different from conventional aspect languages. Notably, aspects usually return to the join point after they executed the advice, as opposed to Larissa, which specifies a different state. Indeed, simply returning to the join point after executing advice is not expressive enough: such a modification can in many cases be expressed modularly with the parallel product in Argos, as we illustrated in Section 3.4.

11.2.3 Semantic Analysis Tools for Larissa

We have presented two semantic analysis tools for Larissa aspects, the first for interference analysis in Chapter 7, and the second in Chapter 8 which allows the combination of design-by-contract and Larissa aspects. Both rely on the formal properties of Larissa, and thus illustrate the advantages of formally-defined languages.

Aspect interference is a problem larger programs where many aspects are applied, especially when aspects are woven sequentially, as we demonstrated on an example. The operator for joint weaving and the aspect interference analysis are thus helpful additions to Larissa. With the contract weaving, it is possible to apply aspects to a specification. This can be done earlier in the development process when an implementation is not yet available. Furthermore, it enables modular verification of aspects, which may allow larger programs to be verified.

Formal analysis tools as the two we presented already exist for other aspect languages: aspect interference analysis is becoming more and more popular since the appearance of the first algorithms [DFS02], and an approach for the modular verification of AspectJ aspects [GK06] also exists. The analysis tools we presented, however, seem more *precise* than existing approaches, in that the interference analysis only detects interferences that really exist, and the contract weaving returns a contract that does not include more programs than necessary. This, for several reasons, is not surprising. First, Argos and Larissa are not Turing complete languages, as most other aspect languages. This makes formal analysis easier. Second, the semantic definition of Larissa and the powerful pointcuts describe the effect the aspect has on a base program quite precisely, depending only on the semantics of the base program. This allows to reason about the effect of the aspect even if the base program is not known.

11.2.4 Aspects for Parallel and Formally-Defined Languages

Larissa has two characteristics that make it interesting from an AOP point of view: it extends a parallel base language, and it is formally defined and designed to be used in the development of safety-critical systems. We discuss the consequences these points had on the design of Larissa, and what conclusions can be drawn for the development of aspect languages for similar languages.

Parallel Languages. Synchronous programs are explicitly structured into parallel units, that can communicate in a very expressive way. Aspects for synchronous languages must crosscut this parallel structure, which calls for different forms of advice. Indeed, we illustrated in Section 3.4 that we can express some typical concerns with the parallel composition in Argos that can only be expressed modularly with aspects in sequential languages. Some of these concerns seem expressible in other parallel languages in the same way as in Argos, and thus aspect languages for parallel languages must offer more powerful advice constructs.

We must distinguish between two kinds of languages with explicit parallelism: those that include both a parallel construct and imperative code, and those that are purely parallel, without any imperative code. The first group includes Argos, where the Mealy automata have an imperative structure. Indeed, Larissa modifies the imperative structure of an automaton, although it models cross-cutting concerns of parallel programs. Other aspect languages for languages of the first kind will probably also modify the imperative structures in the parallel components, but take the parallel structure into account.

The overall structure of Larissa advice may be adaptable to aspect languages for the first kind of parallel languages: interrupt the base program, execute some aspect code, and return to the base program in another state. A problem specific to parallel languages is that their modules are often designed to run forever, and we must thus introduce a notion of termination, as we did for advice programs in Larissa. However, we do not expect that Larissa's way of specifying the return state in the base program can be easily adapted to other languages: executing a trace is a concept that is difficult to understand in languages that are not based on explicit automata.

We expect aspects language for the second kind of parallel languages to look quite different from Larissa. Indeed, as there is no imperative structure, we do not know where to insert advice code. An example for such a language is the synchronous language Lustre [Hal93], which is purely data-flow. There are clearly cross-cutting concerns in Lustre (e.g., making a program reinitializable requires modifying every line of code), but aspect languages that modify the base program in a way different from Larissa are necessary.

Formal Languages. Larissa is not the first aspect language with a formal definition and formal properties. There are many formal calculi for aspect-oriented languages (some are presented in Section 10.3), but their motivations are not the same as ours. Most other approaches have been designed to understand the semantics of aspect weaving better, whereas in the context of synchronous languages, which are used to program safety-critical software, formal definitions and properties are also necessary to enable formal verification. Synchronous languages, as some other formal languages, are also designed to make verification easy, and to guarantee formal properties by construction. This imposes additional constraints on the development of aspect-oriented languages for use in safety-critical environments.

First, having a formally defined language with a connection to verification tools as base language constrains the aspect language. Weaving an aspect must not destroy the formal semantics of the base program or its connection to the verification tools. The semantics of the woven program must be defined in the same terms as the semantics of the base program, and weaving must produce the input format of the verification tools. In Larissa, we guaranteed that by defining aspect weaving as translation in its base languages Argos.

Second, formal languages have particularly high demands for semantic properties, as, in the context of synchronous languages, the preservation of determinism and completeness, or, more generally, the preservation of semantical equivalence between base programs. Aspect-oriented programming is often criticized because it does not preserve this second property, but in general aspect languages sacrifice it to increase their expressiveness.

We believe whether or not aspects should preserve the equivalence depends on their use. If aspects are used to perform non-functional modifications of the base code, e.g. tracing or profiling, they can not preserve the equivalence, as their goal is to distinguish between different implementations. On the other hand, we claim that for functional aspects, the equivalence between programs should be preserved. We believe that aspect languages can preserve the encapsulation and can yet be expressive enough, if they dispose of powerful pointcuts and advice, as e.g. those of Larissa.

11.3 Perspectives

11.3.1 Extension to Integer Variables

An interesting direction for further work is the extension of Larissa to integer variables. This means extending the definition of an automaton to contain a set of internal integer variables, which can be assigned values by the transitions and on which their conditions can depend. Extending Argos to such variables is discussed in [MR01]. When extending Larissa to such automata, we can still select join point transitions with an observer, which may also contain integer variables. These are then added to the program in the product.

Weaving the trace is more complicated, because the conditions of the transitions depend on the values of the integer variables. This does not present problems for *toInit* advice, because we can use the initial values of the variables to execute the trace. For *toCurrent* advice, however, the target state of the advice transition may depend on the current values of the variables. Thus, we must introduce several transitions, and choose the right one depending on the values of the variables.

It would be desirable that aspects continue to preserve semantic equivalence between programs. The most natural choice is to define semantic equivalence as equivalence of traces, and consider integer variables as part of the internal state of the automaton.

However, this means that when an aspect returns to a state of the base program, it must make sure that the variables have values they could have had in this state already in the base program. Otherwise, the behavior of the state is not defined by the semantics of the base program. Thus, we must not allow inserted programs to modify the internal variables of the base program. Furthermore, when executing a trace, we must take into account the assignments of the transitions we take, and set the variables in the target state accordingly. In the same way, *recovery* aspects, on returning to a recovery state, must also set the variables to the values they had when the recovery state was passed the last time.

This is very restricting, and users may want write more powerful aspects than allowed by these restrictions. To allow more expressive aspects, we can only preserve a stricter equivalence by including the integer variables into the interface. Thus, we say that two automata are equivalent if they have the same variables and produce the same traces, even if the input part of a trace includes arbitrary modifications of the variables. This kind of equivalence does not guarantee any encapsulation of the variables, but can be preserved by very invasive aspects.

Once a way to weave aspects into automata with variables is defined, we can try to extend the interference analysis and the contract weaving. We believe that the interference analysis can be adapted without major problems, because it is mainly concerned with join point weaving, which is not changed much by the introduction of variables. Joint weaving works the same way, and checking the join point program for transitions with several join point signals, too.

Extending the contract weaving to variables also seems possible. Transforming an observer with internal integer variables into a non-deterministic automaton can be done without changing the variables. Then, we must adopt the weaving algorithm for non-deterministic automata to integer variables. If we also introduce integer in- and outputs, we cannot represent each possible output by a transition in the non-deterministic automata. However, we believe that this can be addressed by letting transitions non-deterministically emit an output chosen from a set of integers. This is further discussed in Section 8.6. All this remains to be done, however.

11.3.2 Aspect Languages for Other Synchronous Languages

A second interesting field for further work concerns the development of aspect languages for other synchronous languages, including the imperative language Esterel [BG92] and the data-flow language Lustre [Hal93]. We believe that aspects in all synchronous languages can, as Larissa, use a synchronous observer as pointcut. Our examples indicate that observers are expressive enough to describe join points of aspects in reactive systems, and observers have the nice semantic properties they have in Argos also in all the other synchronous languages. Furthermore, they can be expressed in all synchronous languages, and are known to programmers, who use them for specification and verification of programs.

Because synchronous languages are parallel languages, the discussions in Section 11.2.4 about advice in such languages apply here. As discussed there, Lustre is quite different from Argos, because it has no imperative constructs at all, and we thus need a different kind of advice. Esterel, on the other hand, is similar to Argos in that it combines imperative structures in parallel. The basic structure of Larissa advice could thus be adopted. It seems nonetheless difficult to adapt the finite input trace, which is used to design a return state in the base program for *toInit* and *toCurrent* advice. Such a trace has a clear semantics in an automaton that explicitly contains states and transitions. In programming languages without explicit states its semantics would be much harder to understand. Furthermore, it would be nice if the aspects could be given a source transformation semantics, as we did for Larissa. Encoding a trace into Lustre or Esterel, however, would result in bloated code, that would be difficult to understand.

As opposed to *toInit* and *toCurrent* advice, it seems possible to adopt *recovery* advice to other synchronous languages. Because the return states are specified with a observer instead of a trace, it could be defined as source code transformation in various synchronous languages. However, *recovery* advice only treats a special kind of concerns, those where we must revert to a previous state. Thus, other kinds of advice are clearly needed.

11.3.3 Non-Functional Concerns in Reactive Contexts

Finally, this work may make contributions to systems similar to reactive systems, where aspect-oriented programming has so far not been applied. We have identified two domains where we believe that aspects could be used to model non-functional concerns. They concern the modeling of Systems-on-a-chip (SoC) and the modeling and simulation of sensor networks. In both domains, different models with a different degree of detail exists. Typically, developers start with building a functional model of the system, and later add non-functional properties. These non-functional properties are typical cross-cutting concerns, and adding them by hand to the functional model is difficult and error-prone. It would therefore be much easier if they could be modeled with aspects. We discuss an example of such a non-functional concern for each of the domains we identified: the addition of temporal information to models of Systems-on-a-Chip, and the addition of energy consumption to sensor network simulators.

Temporal Information for Abstract Systems-on-a-Chip Models. The development of SoCs is similar to reactive systems, because SoCs have an input/output relation with their environment and are also highly parallel. We have investigated cross-cutting concerns in SystemC [SyC03], a popular description language for SoCs.

SystemC developers usually start by creating a functional *Transaction Level Modeling* (TLM) model [MRR03], which models the main components of the system, and which serves as a reference for software programmers who write software for this system. These models have the advantage that they are much easier to write and much faster to simulate than a complete specification on the Register Transfer Level (RTL). Such functional TLM models, however, do not contain information about the temporal behavior of its components, and thus cannot be used to estimate the performance of the modeled SoC. Furthermore, if the functional properties of the TLM model hold in the SoC also depends on the temporal behavior of the components: bad temporal properties may lead to deadlocks or race conditions in the SoC. E.g., functional TLM models do not describe the capacities of connections between the modules. If the capacity of such a connection is too small, certain components may not be able to communicate, or messages may be lost arbitrarily. Therefore, an intermediate representation is necessary, that allows performance testing and the detection of such errors, but is still easier to write and faster to simulate than the RTL model. It has thus been recently proposed [CMMC08, TCMM07] to add timing information to TLM models.

Adding such timing information is a non-trivial process that involves modifications that are distributed over the entire program, and that is thus a cross-cutting concern. In his master's thesis [Meu07], Quentin Meunier has modeled TLM models in Argos, and has added timing information with Larissa. Although Argos and Larissa are not suited to test performance, they can be used to check if functional properties of a TLM model are preserved when we add timing information. This work has shown that TLM models can be expressed in Argos, that timing information can be added with Larissa, and that we can then check if functional properties are preserved.

Energy Consumption in Sensor Networks. A second setting that is similar to ours and that could profit from aspect oriented programming is the simulation of sensor networks. Sensor networks consist of sensors with attached batteries and radios that are distributed over a large area to measure something, e.g. radiation values. These sensors communicate with one another, and send the measured data to a sink. When simulating such networks, an important question is how long the batteries of the sensors are likely to last. The energy consumption of such a sensor depends on all its modules and also on the global behavior of the system, and is thus a cross-cutting concern, which could profit from aspect-oriented programming.

Modeling the battery duration is quite complicated, because it depends on many factors, e.g. the measured data and the messages it must relay. On the other hand, the state of charge of the battery also influences the (energy consuming) behavior of the sensor, e.g. how often it sends messages. As SoCs, models of sensor networks also have an input/output relation with their environment, and are highly parallel. They can thus be modeled conveniently with synchronous languages, as in the simulators GLONEMO [SMMM06] and Lussensor [MSBV07].

These would be good starting points to experiment with representing the energy consumption of sensor networks with aspects that could profit from the work presented in this document.

11.4 Concluding Remarks

This thesis has investigated aspect-oriented programming in the context of reactive systems. We have concentrated our efforts on developing a small aspect language for the simple synchronous language Argos, instead of investigating aspects in this domain with a broader scope, e.g. trying to invent aspect languages for different reactive languages, or concentrating on more general formal properties. By investigating a small language in depth, we could identify a number of examples and develop some interesting semantic analysis, which we think are best studied in the context of a simple language. We believe that the insights we have gained will prove useful when reasoning about cross-cutting concerns and developing aspect languages in the wider area of formally-defined and parallel systems, such as the modeling of the non-functional properties we discussed in Section 11.3.3.

Bibliography

[AB06] Sven Apel and Don Batory. When to use features and aspects?: a case study. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 59–68, Portland, Oregon, USA, 2006. 5.4

[Ald05] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586 of *LNCS*, pages 144–168. Springer Verlag, Berlin, July 2005. 10.3, 10.4

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, June 2004. ACM Press. 10.2

[AMP] Ample project webpage. `http://www.ample-project.net`. 5.4

[AMS06a] Karine Altisen, Florence Maraninchi, and David Stauch. Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):297–320, 2006. 1, 4.1, 6.1

[AMS06b] Karine Altisen, Florence Maraninchi, and David Stauch. Modular design of man-machine interfaces with Larissa. In *5th International Symposium on Software Composition*, Vienna, Austria, March 2006. 1, 5.1

[And01] James H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 187–209. Springer-Verlag, 2001. 2.3.1, 10.3

[Asp] AspectJ website. `http://www.eclipse.org/aspectj/`. 2.1

[BA04] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development.* Addison-Wesley, 2004. ISBN 0-32-121976-. 3.4, 10.4

[BDCM05] Davide Balzarotti, Antonio Castaldo D'Ursi, Luca Cavallaro, and Mattia Monga. Slicing AspectJ woven code. In Gary T. Leavens, Curtis Clifton, and Ralf Lämmel, editors, *Foundations of Aspect-Oriented Languages*, March 2005. 7.6

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 1, 2.2.1, 3.2.3, 11.3.2

[BJJR04] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer. 2.3.1, 10.3

[BSR03] Don Batory, Jack Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. In *Proc. 25th IEEE Int. Conf. Software Engineering (ICSE)*. IEEE, 2003. 5.4

[CB05] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design.* Addison-Wesley Professional, 2005. 10.1

[CC04] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In Karl Lieberherr, editor, *AOSD-2004*, pages 56–65, March 2004. 2.1, 5.4, 7.6

[CCK03] Andy Clement, Adrian Colyer, and Mik Kersten. Aspect-oriented programming with AJDT. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003. 10.4

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, June 2000. 8.5

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 54–66, January 19–21 2000. 10.2

[CK03] Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *AOSD'03*, pages 50–59, 2003. 2.1

[CL03a] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-15, Iowa State University, Department of Computer Science, December 2003. 8.5

[CL03b] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, March 2003. 10.4

[CL06] Curtis Clifton and Gary T. Leavens. Minimao1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming, Special*

Issue on Foundations of Aspect-Oriented Programming, 63(3):321–374, 2006. 10.3

[CMMC08] Jérôme Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *Design, Automation and Test in Europe (DATE 2008)*, March 2008. To appear. 11.3.3

[CvdBE07a] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola weavr: Aspect orientation and model-driven engineering. *Journal of Object Technology, Special Issue on Aspect-Oriented Modeling*, 6(7):55–88, August 2007. 10.1

[CvdBE07b] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola Weavr: Model weaving in a large industrial context. In *Industry Track of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2007. 10.1

[DBNS06] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proc. of the 5th Int. Conf. on Generative Programming and Component Engineering (GPCE'06)*. ACM Press, October 2006. 10.2

[DFL$^+$05] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 27–38, Chicago, IL, USA, March 2005. ACM Press. 10.2

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don Batory, Charles Consel, and Walid Taha, editors, *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, 2002. Springer-Verlag. 7.1, 7.1, 7.4, 7.6, 10.2, 11.2.3

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Karl Lieberherr, editor, *AOSD-2004*, pages 141–150, March 2004. 7.1, 7.4, 7.6, 7.7, 8.5, 10.2

[DFS05] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005. 10.2

[DL95] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. 10.4

[DSBA05] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In Kris Gybels, Maja D'Hondt, Istvan Nagy, and Remi Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, September 2005. 7.6

[DTK06] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):207–239, 2006. 10.3

[DW06] Daniel S. Dantas and David Walker. Harmless advice. In *Proceedings of the 33th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-06)",*, volume 41, 1 of *SIGPLAN*, pages 383–396. ACM, January 2006. 2.1, 10.3

[DWWW05] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Polyaml: A polymorphic aspect-oriented functional programmming language. In *International Conference on Functional Programming (ICFP)*, pages 306–319. ACM, September 2005. 2.1, 10.3

[Est] The Esterel studio. `http://www.esterel-technologies.com/products/scade-suite/`. 2.2.1

[Fas] Compiler for fast automata. `http://www.lsv.ens-cachan.fr/fast/`. 9.1

[FBT06] Yishai A. Feldman, Ohad Barzilay, and Shmuel Tyszberowicz. Jose: Aspects for design by contract. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 80–89, Pune, India, September 2006. IEEE Computer Society. 8.5

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000. 10.4

[FvS99] Jeroen J.H. Fey and Jan 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. 4.1

[GB03] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 60–69. ACM Press, March 2003. 10.4

[GK06] Max Goldman and Shmuel Katz. Modular generic verification of LTL properties for aspects. In *Foundations of Aspect-Oriented Languages (FOAL)*, March 2006. 11.2.3

[GK07] Max Goldman and Shmuel Katz. Maven: Modular aspect verification. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 3–18, Braga, Portugal, March 2007. Springer. 8.5

[GSS$^+$06] William Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. In *IEEE Software, Special Issue on Aspect-Oriented Programming*, 2006. 10.4

[Hal93] Nicolas Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993. 1, 2.2.1, 9.1, 11.2.4, 11.3.2

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 2.2.1

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe 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*, September 1992. 9.1, 9.2

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology, AMAST'93*, June 1993. 3.3

[Hoa85] Charles A. R. Hoare. *Communication Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985. 2.3.1

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. *Logics and models of concurrent systems, NATO ASI Series*, F-13:477–498, 1985. 2.2

[JBD] JavaBDD library. `http://javabdd.sourceforge.net/`. 9.2

[JCC] The Java compiler compiler tool. `https://javacc.dev.java.net/`. 9.2

[JJR03] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In Luca Cardelli, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427, Geneva, July 2003. Springer-Verlag. 10.3

[Kat93] Shmuel Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993. 10.3

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development I*, 3880:106–134, 2006. 3.4, 8.5, 10.3

[KG06] Jörg Kienzle and Samuel Gélineau. Ao challenge - implementing the acid properties for transactional objects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 202–213, 2006. 2.1

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072, pages 327–353, 2001. 2.1, 2.1

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, 1997. 2.1

[KM05] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005. 10.4

[KO05] Karl Klose and Klaus Ostermann. Back to the future: Pointcuts as predicates over traces. In *Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, 2005. 10.2

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, SE-3(2):125–143, 1977. 3.3

[Läm02] Ralf Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press. 10.3

[Lar] Compiler for Larissa. `http://www-verimag.imag.fr/~stauch/ArgosCompiler/`. 1, 9.1

[LBW04] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, October 2004. 10.2

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, September 2005. 10.2

[LGLL91] Paul LeGuernic, Thierry Gautier, Michel LeBorgne, and Claude LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991. 1, 2.2.1

[LHB05] Roberto E. Lopez-Herrejon and Don Batory. Improving incremental development in AspectJ by bounding quantification. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies*, March 2005. 2.1, 5.4, 7.6

[LSR05] Neil Loughran, Américo Sampaio, and Awais Rashid. From requirements documents to feature models for aspect oriented product line implementation. In *Workshop on MDD in Product Lines (held with MODELS 2005)*, volume 3844 of *Lecture Notes in Computer Science*, pages 262–271. Springer, 2005. 5.4

[Lus] The Lustre tutorial. `http://www-verimag.imag.fr/~raymond/edu/tp.ps.gz`. 8.4

[LWZ06] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming, Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):240–266, 2006. 10.3

[Mar91] Florence Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, oct 1991. 3

[Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. Springer Verlag, LNCS 630, August 1992. 3.5, 6.3.1

[Meu07] Quentin Meunier. Modeling non-functional properties with aspect-oriented programming. Master's thesis, Université Joseph Fourier, Grenoble, June 2007. In french. 11.3.3

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992. 8.1.1

[MG07] Mike Mortensen and Sudipto Ghosh. Refactoring idiomatic exception handling in c++: Throwing and catching exceptions with aspects. In *Industry Track of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2007. 2.1

[Mil80] Robin Milner. A calculus of communicating systems. *Volume 92 of Lecture Notes in Computer Science*, 1980. 2.3.1

[MM04a] Florence Maraninchi and Lionel Morel. Logical-time contracts for reactive embedded components. In *30th EUROMICRO Conference on Component-Based Software Engineering Track, ECBSE'04*, Rennes, France, August 2004. 8.1.1

[MM04b] Florence Maraninchi and Lionel Morel. Logical-time contracts for the development of reactive embedded software. In *30th Euromicro Conference, Component-Based Software Engineering Track (ECBSE)*, Rennes, France, September 2004. 8.1.1

[MR01] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1/3):61–92, 2001. 1, 2.2.1, 3, 11.3.1

[MRR03] Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf. *SystemC Methodologies and Applications*, chapter 2. Kluwer, 2003. 11.3.3

[MSBV07] Florence Maraninchi, Ludovic Samper, Kevin Baradon, and Antoine Vasseur. Lustre as a system modeling language: Lussensor, a case-study with sensor networks. In *Proceedings of Model-driven High-level Programming of Embedded Systems*, Braga, Portugal, March 2007. 11.3.3

[MTY05] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual caml: an aspect-oriented functional language. In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 320–330. ACM, 2005. 2.1

[NS06] Dong Ha Nguyen and Mario Südholt. Vpa-based aspects: Better support for aop over protocols. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 167–176, Pune, India, September 2006. IEEE Computer Society. 10.2

[NS07] Dong Ha Nguyen and Mario Südholt. Property-preserving evolution of components using vpa-based aspects. In *Proc. of the 4th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07) at ECOOP*, July 2007. 10.2

[NSV$^+$06] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed aop using AWED. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2006. ACM Press. 2.1

[OAH$^+$06] Neil Ongkingco, Pavel Avgustinov, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Julian Tibble. Adding open modules to aspectj. In Hidehiko Masuhara and Awais Rashid, editors, *5th International Conference on Aspect-Oriented Software Development (AOSD)*. ACM Press, 2006. 10.4

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005. 10.2, 10.4

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12), 1972. 1

[PDS05] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier. Compar: Ensuring safe around advice composition. In *FMOODS 2005*, volume 3535 of *lncs*, pages 163–178, jan 2005. 7.6

[RC03] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, New York, NY, USA, 2003. ACM Press. 2.1

[RKA06] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained generic aspects. In Gary Leavens, Curtis Clifton, Ralf Lämmel, and Mira Mezini, editors, *Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), AOSD 2006*. Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), in conjunction with Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), March 20-24, 2006, Bonn, Germany, Mar 2006. 10.4

[RRJ07] Pascal Raymond, Yvan Roux, and Erwan Jahier. Specifying and executing reactive scenarios with lutin. In *Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P)*, Braga, Portugal, March 2007. 9.1

[RS03] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press. 2.1

[RWNH98] Pascal Raymond, Daniel Weber, Xavier Nicollin, and Nicolas Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. 9.1

[SAM06] David Stauch, Karine Altisen, and Florence Maraninchi. Interference of Larissa aspects. In *Foundations of Aspect-Oriented Languages (FOAL)*, Bonn, Germany, March 2006. 1, 7.1

[SAM07] David Stauch, Karine Altisen, and Florence Maraninchi. Larissa, un langage d'aspects pour le développement des systèmes réactifs sûrs. In *3ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2007)*, Toulouse, France, March 2007. 8.1.2

[Sca] The Scade suite. `http://www.esterel-technologies.com/products/scade-suite/`. 2.2.1

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000. 10.2

[SDMML03] Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119, Boston, Massachusetts, USA, March 2003. ACM Press. 2.1

[SGS$^+$05] Kevin Sullivan, William Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, September 2005. 10.4

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, 2002. 2.1

[SHU06] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2006. ACM Press. 10.1

[Sip03] Henny Sipma. A formal model for cross-cutting modular transition systems. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL'03)*, March 2003. 10.1

[SK03] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003. 7.6, 10.3

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190, 2002. 2.1

[SMMM06] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. GLONEMO: Global and accurate formal models for the analysis of ad-hoc sensor networks. In *Proceedings of the First International Conference on Integrated Internet Ad hoc and Sensor Networks (InterSense'06)*, Nice, France, May 2006. 11.3.3

[Sta07a] David Stauch. Formal analysis tools for the synchronous aspect language larissa. Submitted to EURASIP Journal on Embedded Systems, Special Issue on Selected Papers from SLA++P 2007 Model-driven High-level Programming of Embedded Systems, 2007. 1, 7.1, 8.1.2

[Sta07b] David Stauch. Modifying contracts with Larissa aspects. In *Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P)*, Electronic Notes in Theoretical Computer Science, Braga, Portugal, March 2007. To appear. 1, 8.1.2

[STJ+06] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhán Clarke, Neil Loughran, and Awais Rashid. Classifying and documenting aspect interactions. In Yvonne Coady, David H. Lorenz, Olaf Spinczyk, and Eric Wohlstadter, editors, *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 23–26, Bonn, Germany, March 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01. 7.6

[Suu] Manual for the Altimax and Vector models. available at the Suunto website. `http://www.suunto.com/`. 5.2

[SyC03] Open SystemC Initiative. *SystemC v2.0.1 Language Reference Manual*, 2003. `http://www.systemc.org/`. 11.3.3

[TCMM07] Claus Traulsen, Jérôme Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007. 11.3.3

[TNI] TNI Software. `http://www.tni-software.com/`. 2.2.1

[Tra] Argos source code for the tram example. `http://www-verimag.imag.fr/~stauch/ArgosCompiler/contracts.html`. 8.4.2

[VBC01] John Viega, Joshua T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 2001. 2.1

[VH03] Matthias Veit and Stephan Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In Mehmet Akşit, editor, *AOSD'03*, pages 140–149, 2003. 5.4

[VSCF05] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful aspects in jasco. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Proceedings of the 4th International Symposium on Software Composition (SC'05)*, pages 167–181, April 2005. 10.2

[Wam06] Dean Wampler. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. In Yvonne Coady, David H. Lorenz, Olaf

Spinczyk, and Eric Wohlstadter, editors, *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, Bonn, Germany, March 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01. 8.5

[WKD04] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transaction on Programming Languages and Systems*, 26(5):890–910, 2004. 10.3

[WV04] Robert J. Walker and Kevin 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*, pages 159–169, New York, NY, USA, 2004. ACM Press. 10.2

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, August 2003. 10.3

ABSTRACT

Title: Larissa, an Aspect-Oriented Language for Reactive Systems

Aspect-oriented programming encapsulates cross-cutting concerns into aspects. Although these concepts have met great success in software engineering, they have never been studied in the context of reactive systems. This thesis takes a first step in that direction. We have developed Larissa, an aspect-oriented extension for the small synchronous programming language Argos. We also studied several examples of cross-cutting concerns in reactive systems, and modeled them with Larissa. Larissa differs from most other aspect languages in two points. First, it crosscuts the *parallel* structure of synchronous languages. Second, it is formally defined and has important semantic properties, notably the preservation of the equivalence of base programs. We also present two analysis tools for Larissa. The first statically analyzes interferences between aspects, and the second combines Larissa aspects with design-by-contract, and thus enables modular verification.

Keywords: Aspect-oriented programming, reactive systems, synchronous languages, formal semantics

RÉSUMÉ

Titre: Larissa, un langage aspect pour les systèmes réactifs

La programmation par aspects encapsule des préoccupations transverses dans des aspects. Alors que ces notions ont eu un grand succès dans le génie logiciel, elles n'ont jamais été étudiées dans le cadre des systèmes réactifs. Cette thèse fait un premier pas dans cette direction. Nous présentons Larissa, un langage d'aspect pour le langage simple synchrone Argos. Nous avons aussi étudié plusieurs exemples de préoccupations transverses dans le domaine des systèmes réactifs, que nous avons modélisés avec Larissa. Les aspects Larissa se distinguent en deux points de la plupart des langages aspects existants. D'abord, ils encapsulent des préoccupations qui sont transverses à la structure parallèle du programme de base. Deuxièmement, ils sont définis formellement et ils ont des propriétés sémantiques importantes, tel que la préservation de l'équivalence entre des programmes de base. La définition sémantique nous a aussi permis de développer deux outils d'analyse puissants, un pour l'interférence des aspects, et l'autre pour la combinaison des aspects avec la programmation par contrat.

Mots-clés: Programmation par aspects, systèmes réactifs, langages synchrones, sémantique formelle