Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2006

CONCURRENT ASPECTS

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Angel Núñez López

Promoter: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Dr. Jacques Noyé and Dr. Rémi Douence (Ecole des Mines de Nantes)

Abstract

Aspect-Oriented Programming (AOP) promises the modularization of so-called crosscuting functionality in large applications. Currently, almost all approaches to AOP provide means for the description of sequential aspects that are to be applied to a sequential base program. A recent approach, Concurrent Event-based AOP (CEAOP), has been introduced, which models the concurrent application of aspects to concurrent base programs. CEAOP uses Finite State Processes (FSP) and their representation as Labeled Transition Systems (LTS) for modeling aspects, base programs and their concurrent composition, thus enabling the use of the Labeled Transition System Analyzer (LTSA) for formal property verification.

CEAOP does not provide an implementation of its concepts, restricting the study of concurrent aspects to the study of a model. The contribution of this thesis is the production of an implementation of CEAOP as a small DSAL (Domain-Specific Aspect Language), Baton, which is very close to FSP, and can be compiled into Java. As an intermediate layer, we have developed a Java library which makes it possible to associate a Java implementation to a finite state process. The compilation process consists of using the Baton program to translate both the Baton aspects and the Java base program into Java finite state processes. This translation relies on Metaborg/SDF to extend Java with Baton and Reflex to instrument the base program.

Acknowledgements

I would like to thank my promotors prof. Dr. Theo D'Hont, Dr. Jacques Noyé and Dr. Rémi Douence for giving me the opportunity to do this thesis. I would like to express my gratitude to Dr. Jacques Noyé for guiding me and helping me in this research until the last moment.

I would like to thank my friends from EMOOSE and EMN that make this time very pleasant.

Thanks to my family for the unconditional affection and emotional help.

Thanks to Anna-Loeke for making this time in France unforgettable.

Table of Contents

1 Introduction **1**

2 State of the Art **3**

2.1 Aspects matching join point sequences . . . 3

2.1.1 Stateful aspects and pattern matching based approaches . . . 3

2.1.2 Logic-based approaches . . . 6

2.2 Modeling concurrency with LTSA . . . 7

2.2.1 Modeling processes . . . 7

2.2.2 Modeling concurrency . . . 10

2.3 Concurrent EAOP . . . 12

2.3.1 Translation . . . 13

2.3.2 Composition . . . 15

3 Toward an implementation of CEAOP **18**

3.1 User point of view . . . 18

3.1.1 The intention and the scenarios . . . 18

3.1.2 Basic hypotheses about the aspects and the base program . . . 19

3.1.3 A concrete aspect syntax for CEAOP . . . 19

3.2 An implementation of LTS composition . . . 25

3.2.1 Introduction . . . 25

3.2.2 Semantics of LTS composition . . . 26

3.2.3 Principles of the synchronization using a centralized monitor . . . 27

3.2.4 Implementation of LTSs as active objects . . . 28

3.2.5 Implementation of the synchronization and the monitor 28
3.2.6 Evaluation 30
3.3 Principles of the implementation 31
3.3.1 Aspects and base program as prefixed LTSs 31
3.3.2 Objects implementing aspects and base program LTSs 33
3.3.3 Instrumenting the base program 33
3.3.4 Operators 34

4 A concrete implementation and optimizations 38
4.1 A Java implementation of the LTS composition 39
4.1.1 The monitor 39
4.1.2 The `LTS` hierarchy 39
4.2 High-level layer: aspects and weaving 41
4.2.1 Implementing aspects 41
4.2.2 Weaving using Reflex 42
4.3 Low-level layer: aspects and base program as LTSs 45
4.3.1 The LTSs modeling the aspects 45
4.3.2 The LTSs modeling the base program 47
4.3.3 Configuring the composition 50
4.4 Implementing the DSAL 51
4.4.1 Overview of Metaborg 51
4.4.2 Aspects 52
4.4.3 Mappings 54
4.4.4 Connectors 55
4.4.5 Implementing aspect binding 56
4.5 Optimizations 57
4.5.1 Avoiding waiting loops in aspects 57
4.5.2 Optimizing the object `Monitorable` 59
4.5.3 Eliminating the double synchronization 59

5 The Readers and Writers problem **60**
5.1 The problem 60
5.2 A solution using CEAOP 60

6 Conclusions **62**
6.1 Contributions 62
6.2 Perspectives 62
6.2.1 Improvements to the implementation 62
6.2.2 Improvements to the CEAOP model 64

Bibliography **66**

List of Figures

2.1 A model of a coffee machine 8
2.2 A model of a simple e-commerce base program 12
2.3 A first model of the LTS of the aspect `Consistency`. 13
2.4 LTS of the aspect `Consistency` where skippable actions have been split up. 14
2.5 LTS of the aspect `Consistency` 14
2.6 LTS of the e-commerce application 15
2.7 The consistency aspect (a) and the safety aspect (a") in FSP. 15
2.8 Woven example 16
2.9 Woven example with more concurrency 16
2.10 Composition of the aspects `Consistency` and `Safety` with the base program using `ParAnd` 17

3.1 Example of synchronization. 28
3.2 LTSs of the threads of a model of an e-commerce application 31
3.3 Prefixed LTSs of the threads of a model of an e-commerce application . . 32
3.4 Example of synchronization of two components using `ParAnd`. 36
3.5 Example of synchronization of three components using `ParAnd`. 37

Chapter 1

Introduction

Building computational systems is about abstracting and modeling some part of the elements or information flows of the world. In this regard, most of the effort in the last time has been the development of methodologies, languages and paradigms that facilitate such a modeling and implementation. Object-oriented Programming and Component-oriented programming are big discoveries in this context. The abstraction of problems using objects and/or components reduces the gap between the subject studied and its modeling. It introduces good properties such as modularization and reuse. However, it fails in the modularization of the so-called crosscuting functionalities which are scattered among the distinct entities of a computer system. Non-functional concerns such as monitoring, debugging, coordination among others are examples of crosscuting functionalities that make the programming less intuitive and less straightforward. They also make objects and components more complex.

Aspect-oriented programming (AOP) [1, 2] promises means for the modularization of such crosscuting functionalities permitting objects and components to get rid of these concerns. The state of the art includes several languages supporting sequential AOP, most notably AspectJ [3]. However these languages support in a very limited way the potential concurrency of their target applications. Some expressions such as the `perthread` in AspectJ make it possible just to apply exclusive aspects to the distinct processes forming an application but without any interaction among them. Therefore, concurrency issues such as mutual exclusion and coordination have to be explicitly treated using libraries for concurrent programming.

A recent approach named Concurrent Event-oriented Programming (CEAOP) [4, 5] has been proposed that provides support for aspects applied over concurrent applications. It is based on the model of Event-based AOP (EAOP) [6] and proposes the modeling of aspects and aspect weaving by a transformation into the calculus of Finite State Processes (FSP) [7]. The FSP semantics models concurrent processes which are composed and ver-

ified with the tool Labeled Transition System Analyzer (LTSA) [7]. Using this tool it is possible to model concurrent applications that are free of deadlocks and data races.

The CEAOP approach allows the coordination of concurrent aspects in a compositional way using high-level operators. These aspects can execute their advices in coordination with the base program using the CEAOP semantics, without requiring the need of a different mechanism (aspect-oriented or otherwise) whose only purpose is to introduce synchronization-related code.

However, CEAOP has been proposed and developed as a model and not much has been done in terms of a concrete implementation. The importance of such an implementation is that it would permit to test and experiment with concrete scenarios, to study the real applicability of the concepts introduced.

Since the approach is based on modeling aspects using FSP and its compositional semantics, a starting point in the generation of an implementation should be the study of how processes described in FSP are implemented. Unfortunately, once processes have been modeled using FSP and tested using LTSA, their implementation still remain to be done "by hand", as a result, there is not certainty that the final product is effectively correct.

The main objective of this thesis was the production of a concrete implementation of CEAOP. With this objective in mind, it was developed a library that translates the composition in FSP of concurrent processes into running Java processes correctly composed. By means of a direct translation, the composition of the implemented processes is guaranteed to the model. Using this implementation as a target, a small DSAL permits the composition of concurrent aspects and put in concrete form the concepts introduced by CEAOP.

This dissertation is structured as follows:

- Chapter 2 reviews the state of the art. First, it exposes the state of the art with respect to stateful aspects, which correspond to the kind of aspects supported by CEAOP. Second, it introduces the main concepts of FSP and LTSA necessary for the understanding of this report. Finally, it presents CEAOP.
- Chapter 3 introduces our implementation of CEAOP. It presents the DSAL developed in this thesis and exposes the principles of its implementation. This chapter also exposes the library for the composition of processes described using FSP.
- Chapter 4 describes the concrete implementation of our DSAL and FSP composition. The result is a running prototype that puts in concrete form the concepts of CEAOP. In addition, the chapter discusses some optimizations.
- Finally, Chapter 5 illustrates the use of our prototype and chapter 6 concludes.

Chapter 2

State of the Art

This chapter exposes the state of the art, that is the starting point of the work developed in this thesis.

Section 2.1 overviews concepts related to stateful aspects. This is useful because the kind of aspects this work considers are basically stateful. Section 2.2 introduces LTSA a methodology for modeling concurrent programs, which is the methodology used in this thesis to model concurrent aspects. Finally, section 2.3 introduces Concurrent EAOP which is the main topic of this thesis.

2.1 Aspects matching join point sequences

A new generation of aspects have appeared with the introduction of stateful aspects [8, 6]. These aspects are defined in terms of sequences of execution points, rather than single join points, *i.e.,* the result of matching its pattern is not any more an isolated program point, but a sequence of join points in the history of computation. In this way, stateful aspects allow us to customize the current computation of a program based not only on the current operation but also on its previous behavior. In other words, they extend and make more complete the initial concept of aspects.

2.1.1 Stateful aspects and pattern matching based approaches

Event-based AOP (EAOP) is the approach used to introduce stateful aspects. It was first documented in [9] as a language with a semantics formally defined by means of parser operators matching event patterns in execution traces. The form of EAOP described in [8, 6] defines crosscuts using regular expressions. Therefore, EAOP can be considered as a

pattern matching based approach. After EAOP several other approaches based on pattern matching have been defined, *e.g.,* JAsCo [10], abc [11], DEP AspectJ [12], among others.

It can be useful to visit some general ideas in the area of pattern matching and to apply them to the stateful aspects approaches. A pattern is a form, template or model (or more abstractly, a set of rules) describing things or parts of things. Pattern matching corresponds to finding (or matching) objects satisfying a pattern in a collection of these objects. A well-known case is the pattern matching of text or sequences of elements. In this case, the collection is a big sequence of elements and the intention is to match subsequences. In terms of pattern matching of sequences, patterns are usually described using regular expressions.

A regular expression describes a set of syntax rules to be applied sequentially over the elements of a collection. The rule that is applied to each element is part of the semantics of the regular expression. For example, if the regular expression $(ad|b)^*c$ is applied over the text `"acaabc"`, for the first letter `"a"` of the text, the following *local rule* is applied: `(match a` $\vee$ `match b` $\vee$ `match c)`. Since the letter `"a"` is matched, for the second letter `"d"` of the text a new rule `(match a` $\vee$ `match b` $\vee$ `match c` $\vee$ `match d)` is applied. When a final rule is matched, then it is possible to say that a sequence has been found. Looking at this process, it is possible to describe the application of a regular expression as a *sequential switch of local rules matching single elements*.

Applying these concepts to the case of aspects, the elements of the sequences correspond to representations of joins points including both static information (*e.g.,* AST fragments) and dynamic information (*e.g.,* variable values). Then, patterns provide rules that constraint both kinds of information at each join point representation. Patterns are applied over the elements of the collection, which corresponds to the trace of computation.

In the following, some aspect approaches based on the pattern matching of sequences are described.

The switch of local rules in the application of a pattern is well-observable in EAOP, which describes an aspect using the following syntax:

$$
\begin{array}{llr}
A ::= & \mu a.A & \textit{recursive definition} \\
 & |\ C \rhd I; A & \textit{prefixing} \\
 & |\ C \rhd I; a & \textit{end of sequence} \\
 & |\ A_1 \square A_2 & \textit{choice}
\end{array}
$$

The basic rule in the formalism of the language is $C \rhd I$, where C is a *crosscut* function that takes a join point (representation) and returns true if the join point matches, false

otherwise (it is a predicate), and I an *insert*. The semantics of this rule is that when the crosscut matches the current join point, it yields a substitution which is applied to the insert before executing it (the substitution takes into account the variable bindings produced in the matching). The meaning of the first rule is that the aspect $\mu a.A$ is the same aspect A where all occurrences of a are replaced by $\mu a.A$. The second rule expresses a sequence of aspects, so that, when the crosscut C matches, then the insert I is performed and the following aspect becomes active. The third rule expresses the end of the sequence and finally, the fourth rule is a choice such that either the aspect A_1 or A_2 are applicable, but when A_1 is applied then A_2 is not.

The combination between both the first and the third rule is the basis of the definition of the kind of event sequences that this approach supports. These two rules implement a tail recursion that allows us to express patterns matching regular languages of events.

In the presence of a choice ($A_1 \Box A_2 \Box ... \Box A_n$), this approach exposes the application of a local rule ($C_1 \lor C_2 \lor \ldots \lor C_n$) matching single join points through the functions C_i such that $A_i = C_i \rhd I_i$. When a join point is matched, the rule is replaced by another rule involving crosscut functions to proceed with the execution of the EAOP aspect.

In terms of stateful aspects, CASB [13] defines a model similar to the model of EAOP, but it details how join point matching is performed, based on its general model of AOP. A local rule to be applied over a join point is modeled as the application of two functions: the function ψ matching static information and the function ϕ matching dynamic information. These functions (ψ and ϕ) are defined as:

$$\psi(i) = (\phi, t, \psi') \quad \textit{and} \;\; \phi(\Sigma) = a$$

where i and Σ correspond to the static and dynamic parts of a join point, respectively. a denotes an advice, and t denotes the kind of aspect (before, after, around,...). The semantics behind this rule is that when the static part of a join point i is matched then the result is a triplet (ϕ, t, ψ'). If the function ϕ matches the dynamic part Σ, then it is possible to say that the pattern has matched a new sequence and the result is the advice a. As a result of the match, the function to be applied to the next join point becomes ψ', which corresponds to a switch of the local rule of the pattern.

JAsCo [10] is a stateful aspect approach that can be considered as an implementation of EAOP. It is based on the explicit programming of a state machine matching sequences. An aspect in JAsCo is programmed defining the transitions of such a machine, where crosscuts associated to each transition are defined using AspectJ pointcuts.

The approaches previously referred are considered stateful aspect approaches because it is always possible to distinguish a state in the aspect. This state defines the local rule to be applied for the next join points to be matched. An important feature of the stateful aspect approaches is that they make it possible to perform interaction analysis, as exposed in [6].

Other kinds of aspect approaches matching sequences have been introduced. They could be considered as stateful aspects but indirectly, since they do not show an explicit state. These approaches directly define their patterns using regular expressions.

The extension to the AspectBench Compiler abc [14] for the AspectJ language, presented in [11], is an approach to stateful aspects that uses regular expressions to define the pattern. A local rule to be applied to a single join point is declared using a *symbol* that maps an AspectJ-like pointcut. The pattern is expressed in a regular expression that uses the defined symbols.

DEP AspectJ [12] is another approach that uses regular expressions. Like the abc approach, some names are associated to AspectJ-like pointcuts that afterward are used in the definition of the pattern. An important feature is that it permits to refer to sequences matched in the past, *i.e.,* to save a kind of history of matched sequences. Furthermore, it permits to match context-free sequences.

2.1.2 Logic-based approaches

Logic-based approaches implement the detection of sequences using logic queries. One of these approaches is ALPHA [15], which corresponds to a pointcut language implemented as an interpreter for a small statically typed AO language. The main characteristics of the approach are the existence of different models of the program semantics that represent on-line databases, and pointcuts that are designed as logic queries written in Prolog in order to retrieve facts from those databases.

The models used as the databases are: the abstract syntax tree (AST), the execution trace, the heap, and the static type assignment. From these models, distinct kinds of information can be brought out.

Alpha provides some basic predicates for querying the execution trace. For example, the predicates `calls(ID, ExprID, Receiver, MethodName, Arg)` and `endcall(ID, CallID, ReturnValue)` allow us to match the events of entering a method call and exiting of it. Each event in the trace acquires a timestamp in the variable `ID` that indicates some precedence in the trace. The predicate `now(ID)` gives the current event ID.

Those basic predicates correspond to the basic *facts* for querying the execution trace. Some rules are provided that are written in function of these facts and also facts defined for the other models. An example of these rules is `cflow(ID1, ID2)` that indicates if an event whose identifier is ID1 is in the cflow of ID2 (note that it can correspond to a fact in the past). The most important feature is the capability of adding more rules of this kind to the language, this it is important since more sophisticated predicates can be given, for example to support context-free grammars for sequences of events.

2.2 Modeling concurrency with LTSA

The design of concurrent systems is a complex task prone to subtle errors. Tools for modeling and analyzing the behavior of concurrent systems are very important in this regard. Furthermore, mechanical or algorithmic verification becomes essential for a correct analysis.

LTSA (Labeled Transition System Analyzer), introduced in [7], is a model-checking tool that allows us to check both desirable and undesirable properties for all possible sequence of events and actions in a concurrent system. It is based on a kind of finites state machines, namely LTSs (Labeled Transition Systems), which have well-defined mathematical properties, facilitating formal analysis and mechanical checking.

Due to the fact that representing state machines graphically severely limits the complexity of problems that can be addressed, LTSA works with a textual representation namely FSP (Finite State Processes). FSP descriptions allows us to describe processes in a straightforward manner. They can be easily translated by LTSA to the equivalent graphical form.

The interest of FSP/LTS is that it provides us with a fairly simple model of concurrency that is well documented (with a good understanding of how to implement models in Java) and supported by LTSA.

This section introduces the concepts around LTSA.

2.2.1 Modeling processes

A process is the execution of a sequential program. The state of a process consists of the values of explicit and implicit variables in the computation of such a program. A process changes this state by the execution of *atomic actions* such as uninterruptible machine instructions that load and store registers. LTSA introduces an abstract model of a process, by ignoring details of state representation and machine instructions. A process is modeled by having a state that is changed through atomic actions. The change from a current state to a next state corresponds to a transition in a graph that is an abstract representation of the program. The result is the modeling of a process using a finite state machine.

The use of finite state machines is very common in several areas of computing and other sciences. They provide a good understanding of the behavior of the subject studied, which has the advantage of being easy to represent graphically. The models introduced by the LTSA tools also use finite state machines in order to study the behavior of computer processes. LTSA uses models based on finite state machines in order to animate and check the behavior of the overall system before it is implemented.

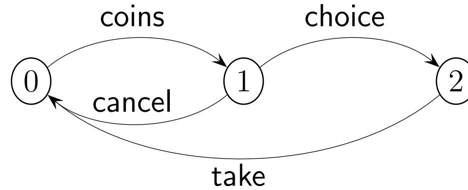


Figure 2.1: A model of a coffee machine

The kind of finite state machines descriptions used by this approach are known as *Labeled Transition Systems* (LTS) because transitions are labeled with action names. Using LTSs it is possible to model processes. For example, the process of a coffee machine could be modeled using the LTS of figure 2.1.

An LTS is a graphical form of a finite state machine description which is excellent for simple processes, but which becomes unmanageable and unreadable for large number of states and transitions. Consequently, LTSA introduced a simple algebraic notation describing an LTS, called FSP. An LTS and an FSP are two representations of the same manner of modeling a process, so that, describing a process using FSP is equivalent to doing it using LTS.

An FSP can be described as follows.

Action Prefix. *If `x` is an action and `P` a process then the action prefix `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as described by `P`.*[1]

This can be shown in the following example:

```
SWITCH = OFF,
OFF = (on -> ON),
ON = (off -> OFF).
```

In this example, the process `OFF` is described as a process that initially engages in the action `on` and then behaves as `ON`. A repetitive behavior is described in FSP using recursion, where the `,` are used to separate processes and the `.` denotes the end of the FSP description. Finally it is possible to apply substitutions obtaining a more compact description:

```
SWITCH = (on -> off -> SWITCH).
```

Choice. *If `x` and `y` are actions then `(x->P | y -> Q)` describes a process which initially engages in either of the actions `x` or `y`. After the first action has occurred, the subsequent behavior is described by `P` if the first action was `x` and `Q` if the first action was `y`.*

A choice is represented as a state with more than one outgoing transition.

[1]This definition and the ones below were taken from [7]

Indexed processes and actions. In order to model processes and actions that can take multiple values, both local processes and action labels may be indexed in FSP. Indexes always take a finite range of values. The process below is a buffer that can contain a single value. It inputs a value in the range 0 to 3 and then output that value.

```
const N = 3
BUFF = (in[i:0..N] -> out[i] -> BUFF).
```

`N` corresponds to the definition of a constant. FSP translates each index into a dot notation `.` for the transition label, so that `in[0]` becomes `in.0`, and so on. The code above is analogous to the code below:

```
BUFF = (in[0] -> out[0] -> BUFF
       |in[1] -> out[1] -> BUFF
       |in[2] -> out[2] -> BUFF
       |in[3] -> out[3] -> BUFF
       ).
```

An equivalent definition, that uses indexed local processes, is shown below:

```
const N = 3
BUFF = (in[i:0..N] -> out[i] -> STORE[i]),
STORE[i:0..N] = out[i] -> BUFF).
```

The scope of a process index variable is the process definition. The scope of an action label index is the choice element in which it occurs. Consequently, the two definitions of the index variable `i` in `BUFF` above do not conflict. On the other hand, both process and action labels may have more than one index.

Indexed processes may induce transitions to undefined states, as shown as follows:

```
const N = 3
COUNT[i:0..N] = (inc -> INC[i+1]).
```

The state `COUNT[3]` defines a transition to a state `COUNT[4]` that does not exist.

FSP defines an *error* state, denoted `ERROR`, which is a terminal state. When an LTS transits to the error state through some action, there is no action that makes the LTS get out of such a state.

Then, the problem can be fixed adding `COUNT[4] = ERROR`. Since the FSP compiler automatically maps undefined states to the error state, this process definition can be omitted.

Guarded Actions *The choice (`when B x -> P | y -> Q`) means that when the guard `B` is true then the actions `x` and `y` are both eligible to be chosen, otherwise if `B` is false then the action `x` cannot be chosen".*

The example below is a process that encapsulates a count variable. The count can be increased by `inc` operations and decreased by `dec` operations. The count is not allowed to exceed `N` or be less than zero.

```
const N = 3
COUNT = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc -> COUNT[i+1]
               |when(i>0) dec -> COUNT[i-1]
               ).
```

Process alphabets. *The alphabet of a process is the set of actions in which it can engage.*

In general the alphabet groups the actions of an FSP that are visible to the other FSPs. Internal actions, labeled `tau`, correspond to actions that are not shared and are invisible to other FSPs. These actions do not belong to the alphabet of the FSP.

As an example, the FSP of figure 2.1 is as follows:

```
COFFEE_MACHINE = (coins->choice->take -> COFFEE_MACHINE
                 |coins->cancel->COFFEE_MACHINE
                 ).
```

2.2.2 Modeling concurrency

The execution of a concurrent program consists of multiple processes active at the same time. In the real life, this is only possible having multiple processors running in parallel, so that, each process can be run in its own designated processor. In this case, it is possible to talk about a *real-concurrent execution.* However, most of the times the amount of processes running are bigger than the number of available processors. In such cases, the processors are switched between processes, so that, they alternate small times of CPU until finishing their execution. Then, it is possible to talk about a *pseudo-concurrent execution.*

Due to the fact that it corresponds to the most general case, the model of concurrency introduced by LTSA includes the case of a pseudo-concurrent execution. Concurrent execution implies then a virtual processor executing a sequence of instructions which is an *interleaving* of the instruction sequences from each individual process. In the most general case, this interleaving is arbitrary, so that there is not any criteria to predict which instruction of which process will be the next to be executed at some time.

2.2.2.1 Parallel composition

FSP provides semantics to model the concurrent execution of actions: *An action `a` is concurrent with another action `b` if a model permits the actions to occur in either the order `a->b` or the order `b->a`.*[2]

[2] This definition and the one below were taken from [7]

Concurrent actions in FSP are the result of the composition of two or more processes using `||` as the designated operator, which is defined as follows: *If `P` and `Q` are processes then `(P||Q)` represents the concurrent execution of `P` and `Q`. The operator `||` is the parallel composition operator.*

The state machine representing the composition generates all possible interleaving of the traces of its constituent processes and is formed by the Cartesian product of them.

2.2.2.2 Shared actions

Interaction between processes is modeled through shared actions. When the alphabets of the composed processes intersect in some actions, these actions are said to be *shared actions*. While unshared actions may be arbitrary interleaved, *a shared action must be executed at the same time by all the processes that participate in that shared action* [7].

This section has introduced the main concepts about modeling of concurrency using LTSA. There are some other elements that are not useful for describing our work. Further details may be seen in [7].

2.2.2.3 Some FSP syntax for concurrency

FSP provides some syntax for customizing the composition of processes, the following features are used in the context of our work.

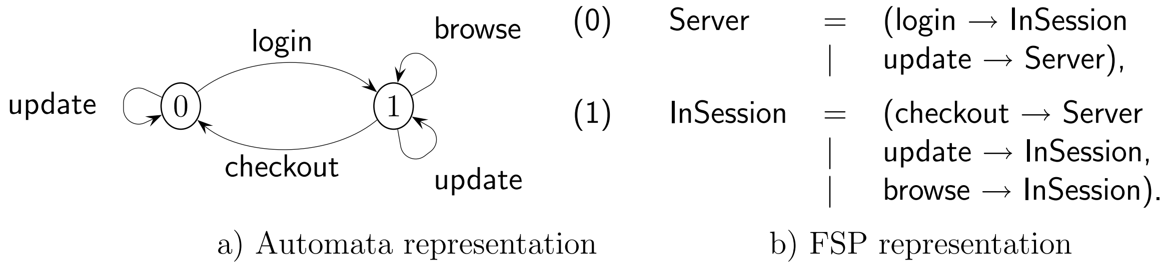
Process prefixing. *`a:P` prefixes each action label in the alphabet of `P` with the label `a`". In an analogous way: "{`a1,..,ax`}`::P` replaces every action label `n` in the alphabet of `P` with the labels `a1.n, ..., ax.n`. Further, every transition `(n->Q)` in the definition of `P` is replaced with the transitions {`a1.n, ..., ax.n`}`->Q`.*

Relabeling. *Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:*
{`newlabel_1/oldlabel_1,...,newlabel_n/oldlabel_n`}.

Hiding. *When applied to a process `P`, the hiding operator `/`{`a1..ax`} removes the action names `a1..ax` from the alphabet of `P` and makes these concealed actions 'silent'. These silent actions are labeled `tau`. Silent actions in different processes are not shared.*

2.3 Concurrent EAOP

Concurrent EAOP (CEAOP) is introduced in [4] and [5]. To illustrate its concepts we will use a running example inspired by typical e-commerce applications. Let us consider the following e-commerce base program. Clients connect to a website and must log in to identify themselves, then they may browse an online catalog. The session ends at checkout, that is, as soon as the client has paid. In addition, an administrator of the shop can update the website at any time by publishing a working version. Figure 2.2 illustrates this example.



a) Automata representation b) FSP representation

Figure 2.2: A model of a simple e-commerce base program

Let us now consider the problem of canceling updates during sessions to the client-specific view of the e-commerce shop, *e.g.,* to ensure consistent pricing to the client. Using EAOP we can define a suitable aspect, called Consistency as follows:

$$\mu a. \big(\mathsf{login}; \mu a'. \big((\mathsf{update} \rhd \mathsf{skip}\ \ \mathsf{log}; a') \,\square\, (\mathsf{checkout}; a)\big)\big)$$

This aspect initially starts in state a and waits for a login event from the base program (other events are just ignored). When the login event occurs, the base program resumes by performing the login, and the aspect proceeds to state a' in which it waits for either an update event or a checkout (other events being ignored). If update occurs first, the associated advice skip log causes the base program to skip the update command (skip is a keyword) and the aspect performs the log command. Then the base program resumes and the aspect returns to state a'. If checkout occurs first, the aspect returns to state a and the base program execution resumes. Since updates are ignored in state a, updates occurring out of a session are performed, while those occurring within sessions (state a') are skipped.

On the other hand, each time the website is updated (*i.e.,* the administrator publishes an internal working version), it is desirable that a second aspect rehashes a database of links before the publication, and backups the database afterward. The second aspect, called Safety, can be defined as follows:

$$\mu a''.(\mathsf{update} \rhd \mathsf{rehash\ proceed\ backup}; a'')$$

Both aspects interact through the action update, and the composition can be determined using EAOP in the sequential case. However, sometimes the concurrent execution of the

aspects together with the base program is desirable (*e.g.,* in the case of the rehashing and backups of the aspect Safety, which are rather time-consuming operations). CEAOP introduces a model to coordinate the concurrent execution of aspects applied to a base program, which consists of two steps: 1) each aspect and the base program are translated to FSP; 2) the concurrent behavior of the aspects applied to the base program is modeled as the parallel composition of those FSPs.

2.3.1 Translation

The translation consists in generating the FSP representation of both aspects and the base program by translating their automata into FSP. This report does not show the formal details of the translation. They are given in [4]. In an informal way, it is possible to say that for the aspect `Consistency` the resulting LTS should have two states, a first state indicating that no client has logged in, and a second state indicating that some client has logged in. In the first state a `login` makes the LTS transit to the second one. In the second state an `update` is a loop that introduces advices, and a `checkout` makes the LTS transit to the first state. The preliminary LTS would be as shown in figure 2.3 (EAOP-like notation for the label `update` is an abuse of notation).

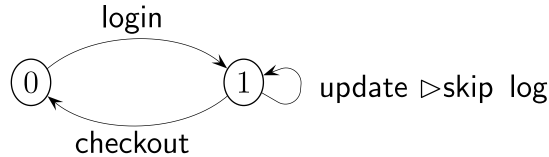


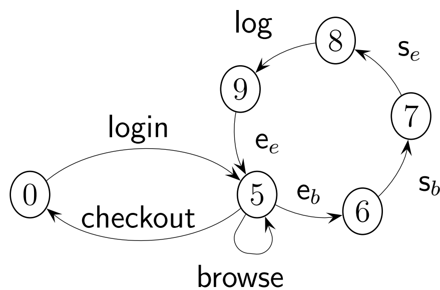
Figure 2.3: A first model of the LTS of the aspect `Consistency`.

The translation introduces synchronization events that will be used to coordinate the aspect and the base program. Aspect expressions of the type $e \rhd \; b \; ps \; a$ are translated to:

$$\text{eventB_}e \rightarrow b \rightarrow \text{eventB_}ps \rightarrow \text{eventE_}ps \rightarrow a \rightarrow \text{eventE_}e$$

where *ps* is one of the keywords proceed or skip, and *b*, *a* denote sequences of actions that are executed respectively before and after *ps*. An action like *e* is called a *skippable action*. By splitting the skippable action update the LTS of the aspect `Consistency` is as follows:

In the translation of aspects, waiting loops are introduced for ignoring events in order to avoid deadlocks. For example, in the first state of the aspect `Consistency` a waiting loop is introduced for the actions `update` and `checkout`, and for the second state another waiting loop for the action `login`. The introduction of waiting loops is done before skippable actions are split, so that waiting loops are also included. The final LTS representing the aspect `Consistency` is shown in figure 2.5.



where the following abbreviations are used:

e_b	=	eventB_update	e_e	=	eventE_update
p_b	=	proceedB_update	p_e	=	proceedE_update
s_b	=	skipB_update	s_e	=	skipE_update

Figure 2.4: LTS of the aspect `Consistency` where skippable actions have been split up.

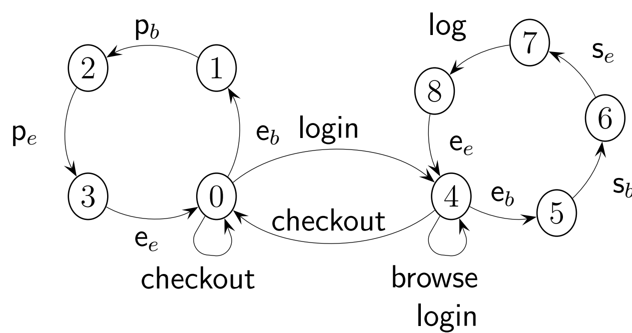


Figure 2.5: LTS of the aspect `Consistency`

Something similar happens with the aspect `Safety`. In the LTS representing the base program, skippable actions are split up by the choice below:

(proceedB_e → e → proceedE_e | skipB_e → skipE_e)

Once skippable actions are split, the LTS representing the base program is shown in the figure 2.6.

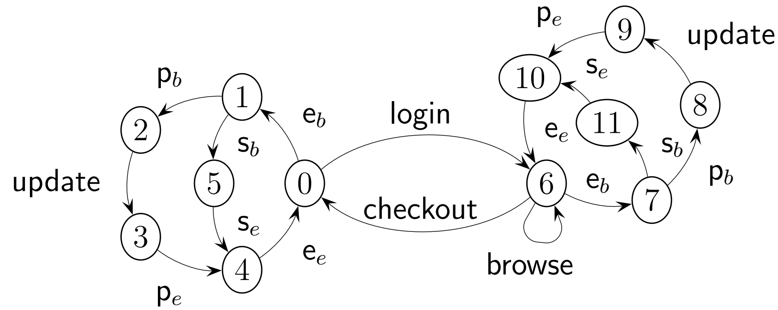


Figure 2.6: LTS of the e-commerce application

Finally, figure 2.7 shows the FSP descriptions that result from the translation of the aspects `Consistency` and `Safety`.

a = (eventB_update → proceedB_update → proceedE_update → eventE_update → a | login → a' | checkout → a | browse → a),

a' =(eventB_update → skipB_update → skipE_update → log → eventE_update → a' | checkout → a | browse → a' | login → a').

a" =(eventB_update → rehash → eventB_proceed → eventE_proceed → backup → eventE_update → a").

Figure 2.7: The consistency aspect (a) and the safety aspect (a") in FSP.

2.3.2 Composition

The parallel composition of FSPs is a form of synchronized product, where interactions are modeled by shared actions. When an action is shared among several processes, the shared actions must be executed at the same time by these processes.

As an example, figure 2.8 shows the output of the composition of the Consistency aspect and the base program. The left-hand side cycle performs updates outside of sessions. The

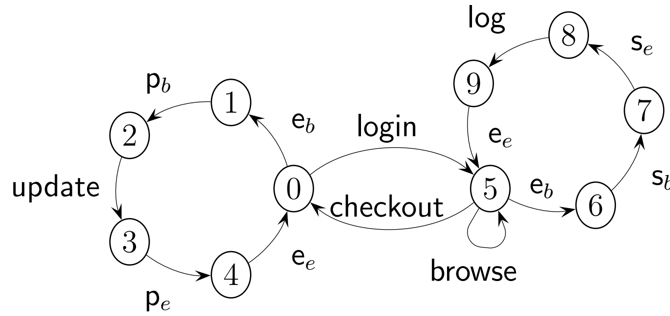


Figure 2.8: Woven example

right-hand side cycle skips update commands during sessions and does some logging. The middle cycle starts and ends sessions.

More concurrency can be introduced by hiding the event `eventE_e` before the parallel composition. The result is shown in figure 2.9. Since after the action `skipE_update` the base program can resume, both interleaving `browse log` and `log browse` are possible.

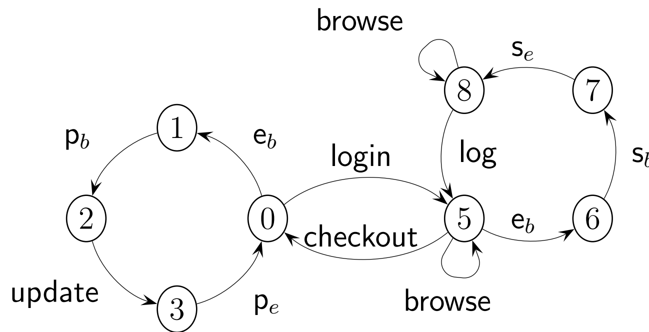


Figure 2.9: Woven example with more concurrency

Composition operators can be designed to compose the aspects in different ways. For instance, let us consider the ParAnd operator. When two advices can be applied at the same join point, their before action sequences are executed in parallel, but there is a rendez-vous on proceed and skip. If both of them wish to proceed, they will proceed in parallel. If (at least) one of them wishes to skip, both will skip in parallel. In our example, ParAnd(Consistency,Safety) composes both advices during sessions to get, using informal syntax, backup skip (log || rehash), which ensures that all database management actions are performed, if reasonable, in parallel.

This composition is modeled in FSP by renaming some synchronization events in the aspect definitions and by defining a process ParAnd that dynamically renames skip and proceed messages. Both aspects share the events eventB_e and eventE_e so that the beginning and the end of advices are synchronized. Before (and after) skip or proceed, advices of the aspects are executed in parallel. The woven program is represented by the automaton of

figure 2.10, where most of the synchronization events have been hidden after the parallel composition (except `eventB_update`, `proceedE_update` and `skipE_update`).

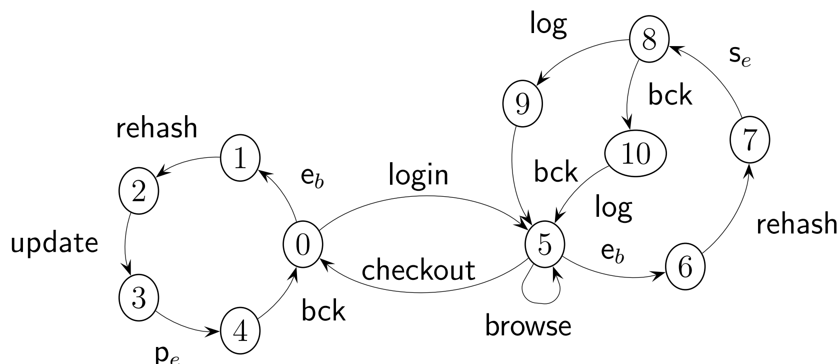


Figure 2.10: Composition of the aspects `Consistency` and `Safety` with the base program using `ParAnd`

It makes clear that the advices are executed in parallel: both sequences log backup and backup log are valid. Furthermore, more concurrency can be introduced by hiding the event `eventE_update`, as a result, after `skipE_update` the base program can resume. We get the same automaton but with loops on `browse` in the states 8, 9 y 10. Then, the user can still browse concurrently with the after advices.

Other operators can be defined similarly. For instance, the advices composed with ParOr proceed when at least one of them proceeds.

More details are available in [4] and [5].

Chapter 3

Toward an implementation of CEAOP

This chapter introduces the main ideas and concepts used to generate an implementation for Concurrent EAOP, which is concretized in the next chapter. Section 3.1 describes the implementation as seen by a user. Section 3.2 exposes a reusable implementation of LTS composition which is afterward used for implementing aspects. Finally section 3.3 explains the principles behind our implementation of CEAOP.

3.1 User point of view

3.1.1 The intention and the scenarios

CEAOP introduces a method for modeling the concurrent execution of aspects together with the base program. It uses LTSA in order to ensure the correct properties of the final concurrent system. That is, once the aspects and the base programs are modeled as LTS, the correctness of the composition can be verified using LTSA.

LTSA allows us to be sure that our final product is correct. However, there is a gap between the model and the final implementation. In spite of the methodologies exposed in [7], which explain how to move from the model to the implementation in Java, the implementation is not direct and there is not a real certainty that they do not introduce problems similar to the ones we are trying to solve (but a different level).

The aim of this thesis is to produce a direct implementation of LTSs and the LTS composition that minimizes as much as possible the gap between the model and the final implementation of concurrent systems. In a next step, this implementation is used to model

concurrent aspects by implementing both aspects and base program threads as LTSs, and by composing them as LTSs.

The importance of providing an implementation for CEAOP is that it makes it possible to experiment and test its applicability in distinct scenarios. In a typical scenario, CEAOP is used to coordinate the parallel execution of the advices of distinct aspects applied over a base program. The work of this thesis allows us to implement this scenario, but also allows us to experiment with another kind of scenarios such as the problem of Readers and Writers shown in the final chapter. The latter is a problem that is solved by our implementation of CEAOP and which uses the power of LTSA complemented with AOP to produce a straightforward solution.

Finally, the aim of this chapter is to produce implementation principles as much independent as possible of a specific base program language, however, some parts are based on the use of Java.

3.1.2 Basic hypotheses about the aspects and the base program

The work developed in this thesis is based on the hypotheses that the CEAOP models state about the aspects. In addition, it has some restrictions about how the base program is provided.

The kind of aspects modeled by this approach are stateful, *i.e.,* at each time during the activity of the aspect, it is possible to recognize an active state with some active transitions. Associated to each transition, it is possible to find a pointcut and some advices. The cut language matches syntactical features of join points. Advices may receive variables bound on pointcuts. Furthermore, each transition is fixed to a keyword either `skip` or `proceed`, indicating if the execution of a matched join point has to be skipped or not.

In terms of the base program, we suppose the availability of either its source code or its bytecode, in order to be able of instrument it using a tool like AspectJ or Reflex.

3.1.3 A concrete aspect syntax for CEAOP

Our implementation of CEAOP is based on the design of a small DSAL (Domain-Specific Aspect Language), called Baton, that allows us to experiment the ideas exposed in this thesis. The composition of Baton with Java is an aspect-oriented extension of Java with concurrent aspects. Baton has been influenced by the languages AspectJ and JAsCo, but introduces some new concepts inherent to the ideas exposed in this work.

The language is divided in three main parts: *aspects*, *mappings* and *connectors*. An aspect

defines an automaton with the transitions of the stateful aspect, using a syntax similar to FSP. We have decided to express the automaton as similar as possible to the syntax of FSP because it facilitates the understanding of our ideas. A mapping allow us to make an association between the labels used in the automaton and either pointcuts or functions. An association *label-pointcut* indicates that a transition is generated by the execution of a join point in the base program. An association *label-function* indicates a routine to be executed together with a transition which corresponds to an advice. Finally, a connector allows to instantiate aspects connecting Baton's aspects to Baton's mappings.

These three parts allow us to make a separation among four concepts of AOP and stateful aspects, permitting the study of them in an independent way:

1. The *join point model and pointcuts* (also referred as the *cut language*) indicating which are the supported join points and where to instrument the base program. It is declared through the pointcuts of a Baton mapping.

2. The *advices* built of pure Java classes. It is declared through the associations between labels denoting advices and functions in a Baton mapping.

3. The *automaton* describing the state machine of the stateful aspect. It is declared in the definition of a Baton aspect.

4. The *instantiation* permitting to create and compose several aspect instances in a Baton connector.

Most of the stateful aspects approaches declares the automaton, pointcuts and advice code in a single module, this permits more encapsulation but impedes the total reuse of these ingredients. A step in reuse is introduced by JAsCo through the declaration of *connectors*.

In our case, we have put more attention in the reuse by a total separation of the conceptual parts involving AOP. However, this is not the only reason. Since our approach is based on the use of LTSA, the automaton is described and composed using the LTS semantics, which put a lot of dependency on the labels used to declare the aspects. In order, to compose aspects, the labels must have the same semantics, *i.e.,* they have to refer to the same pointcuts and/or functions. The separation of the cut language using mappings allow us to declare composition of aspects and instantiate them using a unique semantics for labels.

In terms of encapsulation, not everything is lost with this separation. The definition of the automaton can be seen as a component whose interface are the events indicated by the labels, in an analogous ways as an LTS in [7] is seen as a component. In this component the labels have a fixed role, they can be either input events (pointcuts) or output events (advice routines). In the same way, a mapping can be seen as a component instrumenting the base program, generating events and receiving events in order to execute routines. Then, a connector is able to connect both components in a proper way. The component that

results from the connection of aspects and mappings, has input events that correspond to pointcuts and output events that correspond to the execution of the routines corresponding to advices. In this way aspects are seen as black boxes connected with a base program and executing advice routines when required.

3.1.3.1 Aspects

The declaration of Baton's aspects corresponds to the definition of the automaton of a stateful aspect, as exemplified in the code below:

```
1  aspect Consistency {
2
3    p1 = update -> p1
4       | login -> p2
5       | checkout -> p1,
6
7    p2 = update(admin) > skip; log(admin) -> p2
8       | login -> p2
9       | checkout -> p1
10 }
```

This code describes an aspect as a process `p1` with a subprocess `p2`, and the set of labels {`update, login, checkout`} as its alphabet. The aspect is written using a syntax similar to FSP, with the only exception of the expression `update > skip; log(admin)` on line 7. This expression declares an *extension* of the label `update` with the sequence of labels `skip;log`, which is explained in the remainder. In addition, the label `update` declares a parameter which is passed to the action `log`. This corresponds to the syntax of bindings in pointcuts, also explained in the remainder.

The language also supports the definition of indexed processes. This allows us to declare the previous example by using only one process as follows:

```
1  aspect Consistency {
2
3    p[logged:0..1] =
4       when (logged == 0)
5         update -> p[logged]
6       | when (logged > 0)
7           update(admin) > skip; log(admin) -> p[logged]
8       | login -> p[1]
9       | checkout -> p[0]
10
11 }
```

This example declares the same automaton. The condition `when` allows us to indicate whether a transition is available or not.

An aspect is defined by the following abstract grammar:

$$
\begin{aligned}
aspect &::= id\ automaton \\
automaton &::= process^{+} \\
process &::= id\ paramDecl^{*}\ choice \\
paramDecl &::= param\ number\ number \\
choice &::= sequence^{+} \\
sequence &::= [condition]\ transition^{+}\ id\ paramRef^{*} \\
paramRef &::= param \mid param\ op\ number \mid number \mid number\ op\ number \\
condition &::= param\ comp\ number \\
transition &::= label \mid extension
\end{aligned}
$$

An aspect consists of an identifier and an automaton described, in turn, by one or more processes. A process is described by an identifier, zero or more variable declarations, and a choice. A variable declaration allows us to declare a variable, specifying a range between two numbers. A choice is either a single sequence or a set of sequences. A sequence consists of an optional condition, a sequence of transitions, and the identifier of some process with optional variable references. A variable reference is either a single variable, an arithmetic operation between a variable and a number, a single number or an arithmetic operation between two numbers. A condition is a binary comparison between a variable and a number (this could be easily extended to allow comparing variables as well as with conjunctions, disjunctions and negation). Finally, a transition may be an atomic label or an extension, which is explained straight after.

In general, labels are associated to join points in the base program using a mapping label-pointcut. In this way, the automaton can be connected with the base program. An extension is the way to introduce advices in this process. The syntax of an extension is described by the following grammar:

$$
\begin{aligned}
extension &::= event\ call^{*}\ ps\ call^{*} \\
ps &::= \texttt{proceed} \mid \texttt{skip} \\
event &::= label\ param^{*} \\
call &::= (label\ \mid \texttt{+}label)\ param^{*}
\end{aligned}
$$

where both *label* and *param* are identifiers.

The semantics of this expression describes the replacement of a target label by a sequence consisting of before labels, either the label `proceed` or `skip` and after labels. The target label may declare parameters representing the binding of variables in the base program which may be distributed in parameters among the before and after labels. These parameters become the values passed to advice functions associated to the latter (using a mapping label-function). For example, in our case the line 7 of the previous code declares an extension of the label `update` that is defined by the sequence `skip;log`. In this definition, a variable `admin`, declared as the parameter of `update`, is given as parameter to the label `log`.

In general, the labels associated to advices are considered internal labels, *i.e.,* labels that are not visible to other LTSs, so that, they do not participate to the LTS composition. The last rule of the grammar of extensions allows us to define advice labels that are visible to other LTSs. This is done by prefixing such advices by the sign `+`.

3.1.3.2 Mappings

Mappings allow us to complete the semantics of Baton's aspects by associating labels with pointcuts. In this way, aspects can relate to the base program. Mapping are also used to map the labels introduced by extensions with executable functions. These labels denote advices, so that, the association with functions allows us to perform advice routines.

The pointcuts defined in a mapping are declared using AspectJ syntax, where the name of the pointcut corresponds to the label associated to it. In this preliminary version, the AspectJ expressions provided are only `execution`, `call` and the binding expression `target`.

The definition of functions is simply done by indicating the label and the signature of the corresponding method.

The following code shows an example of the declaration of a mapping:

```
1  mapping MyMapping {
2    pointcut update(Admin param) :
3                execution(public void Admin.update(..)) && target(param);
4    pointcut login() : execution(* *.login(..));
5    pointcut checkout() : execution(* *.checkout(..));
6    function log : Logger.log(Admin);
7  }
```

This code maps the label `update` to a pointcut matching the execution of the method `Admin.update()` and also declares a binding of the object executing the method. The labels `login` and `checkout` declare pointcuts matching the execution of any method

named `login` and `checkout`, respectively. Line 6 maps the label `log` to the Java method `Logger.log(Admin)`.

The abstract grammar describing a mapping is as follows:

$$\begin{aligned} mapping &::= id\ (pointcut \mid function)^* \\ pointcut &::= label\ param^*\ pExpression^+ \\ function &::= label\ methodSignature \end{aligned}$$

A mapping consists of an identifier and zero or more pointcuts or functions. A pointcut is declared using the corresponding label, some optional parameters binding variables, and one or more pointcut expressions. These expressions are the ones taken from AspectJ, such that, `call` and `execution`.

The simplified grammar of the expressions supported in AspectJ is as follows:

$$\begin{aligned} pExpression &::= (\texttt{call} \mid \texttt{execution})\ methodPattern \\ methodPattern &::= MethodMod\ Type\ ClassMemberName\ FormalParameter^* \end{aligned}$$

The pointcut expression can be either a `call` or a `execution` (also it is supported `target`, but it is not detailed) and a method pattern. A method pattern consists of method modifiers, a return type, a class name and zero or more formal parameters.

A function declares just a label and a method signature in Java.

3.1.3.3 Connectors

Connectors are used to instantiate aspects by connecting mappings with aspect declarations as shown in the following code:

```
1 connector MyConnector {
2   Aspect aspect = new Consistency() using MyMapping;
3 }
```

In this code a new aspect is instantiated by connecting the declaration of the aspect `Consistency` with the mapping `MyMapping`. The result is an aspect as an instance of the Java class `Aspect`.

It is also possible to declare aspect instances using operators:

```
1 connector MyConnector {
2   Aspect aspect = new ParAnd(new Consistency(), new Safety()) using MyMapping;
3 }
```

In this case, a new aspect is instantiated using the operator `ParAnd` which receives two aspect instances of both aspect declarations `Consistency` and `Safety`. The three created instances share the same mapping `MyMapping`, which is important in order perform the composition using the same semantics of labels.

Connectors are defined using the following grammar:

$$\begin{aligned} connector &::= id\ aspectInst^* \\ aspectInst &::= aspectComp\ mappingId \\ aspectComp &::= id \\ aspectComp &::= operator\ aspectComp\ aspectComp \end{aligned}$$

A connector consists of an identifier and zero or more aspect instantiations. An aspect instantiation, in turn, is made of a compositional expression relating aspect declarations and a mapping. This expression constitutes a tree of aspects composed using operators.

3.2 An implementation of LTS composition

3.2.1 Introduction

If two or more single processes are running at the same time, it is possible to say that a global process is running, which is the composition of these single processes. Then, each state of the composite process can be mapped to a tuple consisting of one state for each of the component processes. In a schema where processes run independently, any combination of states can be a state of the composition. The semantics of the LTS composition, exposed in this section, describes the composition of processes, where some coordination constraints are imposed, so that not all the state tuples are reachable.

LTS composition consists in the construction of a single LTS describing the composition, which is the combination of the LTSs of the processes to be composed. The semantics of this composite LTS allows us to direct the execution of the composed processes. In a scheme without any synchronization, each single process runs on its own, following its LTS description and taking its own decisions. The LTS describing the composition imposes

synchronization among the single processes, by telling them which transitions to follow at each time.

It is possible to make a metaphor with an Orchestra, such that each LTS can be seen as a written piece of music for a particular instrument and the process running the LTS, as a musician. Then, an LTS describing the composition would correspond to a musical arrangement, which tells how a musical composition has to be played by the different participants of the orchestra.

In contemporary orchestras, there exists a musician playing the role of *conductor*. In general, the conductor is responsible for ensuring entries are made at the right time and that there is a unified beat. In the case of composing LTSs, we have a similar necessity. Some entity having the global view of the execution of the processes should conduct the synchronization of them. We have called this entity a *monitor*, which is in charge of this coordination. Since each process cannot take on its own the decision about which transition to follow at the right time, the decision is taken by the monitor, based on the information obtained from the other processes. Then, the monitor acts as an object dictating to the single processes the transitions to follow at each time.

In the remainder, section 3.2.2 presents the semantics of the LTS composition. This semantics is taken from [7] and expressed in a way that is considered clearer for the purposes of this report. Section 3.2.3 introduces in a declarative form our ideas of synchronization in order to implement LTS composition. Sections 3.2.4 and 3.2.5 introduce an implementation of the LTS composition. Finally, section 3.2.6 presents an evaluation of our ideas.

3.2.2 Semantics of LTS composition

The LTS composition consists in the creation of a single LTS describing the composition of several processes described as LTSs. The semantics of an LTS and LTS composition are presented in [7]. In this section, the ideas behind this semantics are explained in a way more useful for the purposes of this report. We present how the LTS describing the composition is constructed and also what is the semantics behind it.

The following defines how the LTS describing the composition is constructed for the case of n LTSs.

Let us consider n LTSs P_i (with $1 \leq i \leq n$), with their shared alphabets αP_i and their sets of transitions Δ_i. A state s_c of the composition corresponds to a state tuple $(s_1, s_2, ..., s_n)$, where s_i is a state of P_i.

The transition $((s_1, s_2, ..., s_n), a_j, (s'_1, s'_2, ..., s'_n))$ is a transition in the composition, if $\forall P_i$ such that $a_j \in \alpha Pi$, $(s_i, a_j, s'_i) \in \Delta_i$. Then, $\forall i$ such that $a_j \notin \alpha P_i$, $s'_i = s_i$.

Finally, the first state of the composition corresponds to the state tuple $(s_1^0, s_2^0, ..., s_n^0)$, where s_i^0 corresponds to the initial state of P_i.

In other words, in a state $(s_1, s_2, ..., s_n)$ of the composition, a_j corresponds to a transition if $\forall P_i$ such that $a_j \in \alpha P_i$, a_j is a transition in the state s_i. Then the semantics of the composition tell us that all these LTSs transit together through a_j.

3.2.3 Principles of the synchronization using a centralized monitor

Section 3.2.2 presented the semantics of the LTS composition. This section explains the mechanism to compose running processes in order for them to follow this semantics.

The scenario consists of several processes described using LTSs and a monitor conducting their executions. As section 3.2.1 explained, the monitor is necessary in order to decide which transitions the LTSs should follow at each instant. An LTS cannot take this decision on its own, so that it has to wait for such a decision from the monitor. In the remainder, our mechanism of synchronization is described in a declarative way.

Let us consider n LTSs P_i (with $1 \leq i \leq n$), with their shared alphabets αP_i, and their current states s_i. We say that P_i is in state s_i^w if it is in a state s_i and it is *waiting* to pass to a next state, or it is in state s_i^b if it is in a state s_i and it is *busy*. Now, let us consider m possible shared actions a_j (with $1 \leq j \leq m$) such that $\exists i$ such that $a_j \in \alpha P_i$. We define c_j a counter indicating the amount of P_i such that $a_j \in \alpha P_i$ and P_i is in state s_i^w, and b_j a bound indicating the amount of P_i such that $a_j \in \alpha P_i$. At each time, c_j tells us the number of LTSs waiting to pass to a next state using the action a_j.

The monitor is an entity in charge of checking at each time for all the actions a_j whether $c_j = b_j$.

When for an action a_k, $c_k = b_k$, the action a_k may be chosen as an action to proceed. If a_k is chosen, then all P_i, such that $a_k \in \alpha P_i$, transit together to a next state using a_k.

The election of an action a_j such that $c_j = b_j$ is correct with respect to LTS composition. The rationale behind this statement is that in a state $(s_1, s_2, ..., s_n)$ of the LTS composition, the transition labeled a_j effectively corresponds to a transition in the composition, because $\forall P_i$ such as $a_j \in \alpha P_i$, P_i is waiting to pass to the next state using a_j, and therefore a_j is a transition in the state s_i of each of these LTSs.

Figure 3.4 shows an example of the schema of synchronization. We have the LTSs P_1, P_2 and P_3 with the alphabets $\alpha P_1 = \{a_1, a_2\}$, $\alpha P_2 = \{a_2, a_3\}$ and $\alpha P_3 = \{a_2\}$. At a certain instant, P_1 and P_3 are waiting to pass to a next state, whereas P_2 is still busy. The monitor keeps the counters updated and checks that the action a_1 has reached its bound.

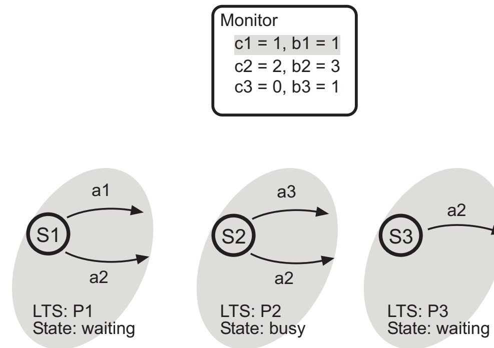


Figure 3.1: Example of synchronization.

3.2.4 Implementation of LTSs as active objects

An LTS is the description of a computation, such that its transitions are associated to the actions performed in such a computation and its states are associated to the different configurations the computation moves between. In this way, the LTS that corresponds to a computation may be deduced by observing its execution. The approach is the other way around when we first have the LTS describing a computation and we want to generate a computation following this description.

In order to generate the computation that follows a given LTS, we have chosen an interpreted schema. The idea is to generate the computation that results from the interpretation of the LTS. This interpretation is the evaluation of the transition used to pass from a state to another, each time a transition is performed. This evaluation results in the execution of a routine associated to the corresponding transition and the passing to the next state in the LTS.

Since we need to be able to compose several LTSs and execute them concurrently, a natural manner to implement them is by using active objects. In this way, an LTS corresponds to an object consisting of (1) its states, (2) its transitions, (3) the routines associated to the transitions (4) and an internal thread where the transitions and the routines are performed. Also we provide this object with a function to determine the next transition to be evaluated and an evaluation function.

The election of the next transition depends on the *monitor*, which is in charge of choosing it in terms of the possible composition with other LTSs.

3.2.5 Implementation of the synchronization and the monitor

The monitor is designed as an object which keeps the variables c_j and b_j described in section 3.2.3, and which is able to choose the next action to be followed by the LTSs in the

composition. It keeps watch over the execution of the LTSs and indicates the transition to follow at the right time to the waiting LTSs. In order to calculate the bounds of each action, the initialization of the monitor requires the complete alphabet of the LTS to be composed.

The mechanism of synchronization can be explained as follows. When an LTS is in some state, ready to pass to a next one, it notifies to the monitor the different actions that it could follow and then it waits for a decision. The monitor increments the counters c_j associated to the actions passed by the LTS, and checks whether some action has reached its bound b_j. If no action has reached its bound, it does nothing. Otherwise, if an action reaches its bound, the monitor chooses the action to proceed with. Then, it wakes up and notifies the decision to all the LTSs having the chosen action in their alphabets. When it notifies the decision to an LTS, it also decrement the counters of the actions not chosen that were previously notified by the LTS . When the LTSs wake up and receive the decision, they transit to the next state. Then, some extra synchronization is necessary to ensure that these LTSs transit together.

3.2.5.1 Interpretation of a shared action

Suppose the monitor has chosen an action a, and the LTSs P_1, P_2,..., P_m are ready to pass to their next states using this action. The LTS semantics says that the action a has to be executed at the same time by all these LTSs. In order to give a general meaning to this definition, the action a can be refined in the *sub-actions* a_1, a_2,..., a_m, where a_i corresponds to the action a in P_i. The execution (or interpretation) of the action a corresponds to the execution (or interpretation) of each sub-action a_i by its LTS P_i. The action a is executed at the same time by all the LTSs P_i if their sub-actions a_i are executed together, *i.e.,* any possible interleaving $b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_m$ is allowed such that $\forall\ b_i\ \exists\ a_i$ such that $b_i = a_i$ (no action is executed in the middle of two sub-actions).

A *synchronization on the entry* of a makes all the LTSs P_i meet at the same point. From this point they begin the execution of their sub-actions a_i, generating some interleaving. A *synchronization on the exit* imposes that all the LTSs meet at the end of the execution of their sub-actions. This impedes that some LTS P_i executes some other action c (such that $\forall\ a_i,\ a_i \neq\ c_i$) when it has finished the execution of its sub-action but some LTS has not already finished. In this way, an interleaving $b_1 \rightarrow b_2 \rightarrow ... \rightarrow c \rightarrow ... \rightarrow b_m$ is avoided.

We can distinguish the necessity of two points of synchronization: the entry and the exit to a transition. These points are described as follows.

3.2.5.2 Synchronization on entry

The synchronization on entry is implemented as the notification that an LTS does when it is ready to pass to a next state. Then, it passes to the monitor the different actions that it could follow. Since to choose an action a the monitor needs all the LTSs having a in their alphabets to notify the synchronization on entry, they do meet before performing the action (if the action is chosen). In an ideal scenario, if the next action to be chosen is known, all the LTSs having this action in their alphabets should wait in a common *waiting set*, so that, through a single notification, they could wake up together. However, the action to be chosen is not known a priori, so it is not possible to group the LTS by any criteria. The better option in this case is not to make any group. Then, the LTSs should wait in their own waiting set. Afterward, the monitor has to wake up these LTSs one by one.

3.2.5.3 Synchronization on exit

When a group of LTSs are transiting together using an action chosen by the monitor, it is necessary to ensure that all of them have exited of the transition before continuing their respective computations. In order to do this, each of them has to notify the monitor of its exit and wait until all the rest have exited. This is done using a *synchronizer* object, which receives notifications from the LTSs and make them wait in a common waiting set. When the last of them has exited, all the rest are woken up in order to continue with their executions.

3.2.6 Evaluation

A big problem of this implementation is that it is too centralized. The decision is always taken by a central entity: the monitor. All the LTSs have to do a notification to this monitor and wait for its decision. This is a problem specially in the case of distributed systems. Since the decisions are taken by a central monitor, the distribution of the LTSs in different hosts requires all the LTSs to establish a remote connection with the monitor which resides in some host. Unfortunately, not much can be done in this regard. Necessarily the schema using LTS composition is centralized by definition. If we come back to our metaphor of an Orchestra, necessarily there to be a conductor. It may also depend on some specific properties of composition (*e.g.,* binary interactions, *i.e.,* shared actions are only shared by two LTSs). This requires further investigations.

3.3 Principles of the implementation

This section introduces the principles of our implementation of CEAOP, which is the basis of the concrete implementation of the next chapter.

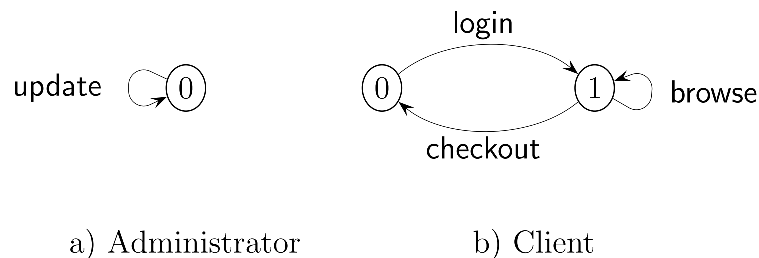
3.3.1 Aspects and base program as prefixed LTSs

The main idea of CEAOP, as described in section 2.3, is to model both aspects and base program as LTSs in order to compose them following the semantics of the LTS composition.

The modeling of aspects and base program as LTSs is explained in 2.3.1. The main idea is that the automaton of an aspect is represented as an LTS by splitting the actions that declare advices (skippable actions) in several synchronization events and by introducing the advices as explicit actions in the automaton. The base program is modeled in a similar way by splitting skippable actions. In addition, in order to avoid deadlocks, the definition of the LTSs representing aspects includes waiting loops.

Section 2.3 models the base program as a single LTS, no matter if such a base program is composed of more than one thread. For instance, the example of the e-commerce application of section 2.3 consists of at least two distinct threads, one representing clients performing the actions `login` and `checkout` (for simplicity the action `browse` is omitted), and a second representing administrators doing updates. The LTS modeling this application is the single LTS of Figure 2.2.

The implementation proposed by this thesis makes the hypothesis that each base program thread is modeled by a distinct LTS. In this way, the e-commerce application is modeled by the LTSs of Figure 3.2 (omiting the action `browse`).



a) Administrator b) Client

Figure 3.2: LTSs of the threads of a model of an e-commerce application

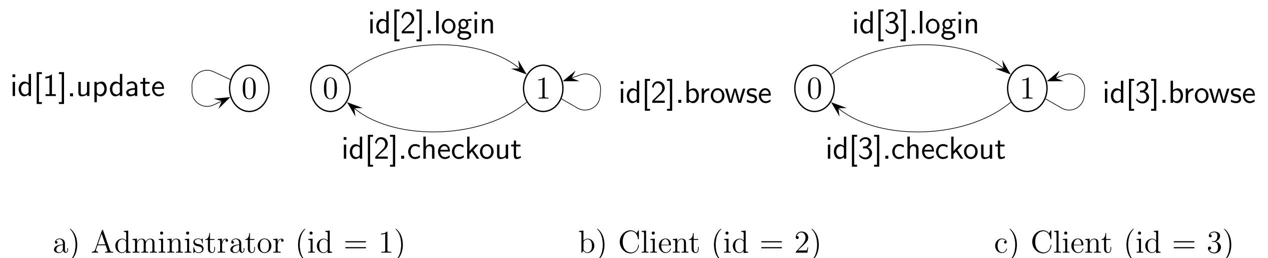
The final woven e-commerce application is modeled by the composition of the LTSs describing the threads of the base program and the ones describing the aspects.

When the base program is a composition of various identical (FSP) processes, care has to be taken to understand the meaning of each individual action. Are there actions shared

between all the base processes or are they only shared with the aspects? In the second case, a different process instance has to be created for each process. For instance, if the e-commerce application executes two threads representing two different clients, then the LTS of clients is instantiated twice. This implies the existence of two LTSs exposing the same alphabet. The composition in the final weaved application has the undesirable result that all their actions are shared and therefore synchronized, *e.g.,* both clients will log in at the same time.

At the description level, this problem can be solved by adding waiting loops. If in the state 1 of the client's LTS of figure 3.2 a waiting loop is added on the label `login`, then a client may log in first and the second afterward by synchronization on the waiting loop of the first client. However, this solution cannot be implemented in the base program. Once the client log in, the base program continues its normal execution and the waiting loop is not available.

The solution of this problem is to generate an unique prefixed instance of the LTS for each thread in the base program. This instance is the result of applying the prefixing `id[n]::P`, where `P` corresponds to the LTS to instantiate and `n` is an unique thread's identifier. The instance exposes the same LTS but with all its transitions prefixed by the identifier. In this way, each thread in the base program is modeled by an LTS having an unique alphabet. In the example, the result of prefixing the LTS for each client thread is shown in figure 3.3, assuming the existence of two clients and one administrator.



a) Administrator (id = 1) b) Client (id = 2) c) Client (id = 3)

Figure 3.3: Prefixed LTSs of the threads of a model of an e-commerce application

Since the LTSs of the base program have been prefixed, the aspect LTSs have also to be prefixed. The prefixing is done before splitting skippable actions. An aspect is prefixed by the expression `id[0..N]::P`, where `N` is the number of threads in the base program and `P` is the LTS of the aspect without splitting skippable actions. The result is a prefixed instance of the LTS that for each transition labeled `a` defines similar transitions labeled `id[0].a`, `id[1].a`,..., `id[N].a`. Afterward, each skippable action, now prefixed, is splitted in the synchronization events described in section 2.3.1.

The solution has the disadvantage that it is limited to a maximum of `N` threads in the base program, which is fixed. However, it exposes a straightforward solution that applies concepts inherent to LTSA. Therefore, this thesis adopts this solution as the manner to model aspects and the threads of the base program. This should be only for the case of

multiple similar threads instances. However, knowing when the base program will create multiple similar threads is not straightforward. Since using prefixing is more general, we adopt this solution.

3.3.2 Objects implementing aspects and base program LTSs

The principle behind the implementation of aspects corresponds to using the concepts introduced in section 3.2.4. An aspect is implemented as an active object interpreting the LTS that models it. Such an object is equipped with some routines to be executed in the interpretation of the transition denoting advices. The other transitions have built-in routines used to impose synchronization, although these routines are not seen by the user.

The concrete execution of an aspect is given not only by the interpretation of its LTS but also by the composition with other LTSs, specially with the LTSs representing the threads of the base program. The composition makes the aspect move between states in coordination with the base program.

The LTSs describing the threads of the base program are implemented using objects called *monitorable* which, unlike the objects implementing aspects, do not interpret their LTSs, the transitions are triggered by the execution of the code that instruments join points in its corresponding base program thread.

Since, the interpreted LTSs implementing aspects and the objects monitorable are both implementations of LTSs, they can be composed using the *monitor* described in section 3.2.5.

3.3.3 Instrumenting the base program

The monitorable object of a base program thread allows the monitor to see such a thread as an LTS and to compose it with the other LTSs. By instrumenting join points in the base program, the control flow is passed to the monitorable object associated to the current thread, which makes the composition possible.

For instrumentation purposes, a group of labels classified into skippable and non-skippable is given. These classified labels are retrieved from the aspects by analysis of their automatons. Skippable labels are ones defining advices. For each label, it is selected a set of join points in the source of the base program (we suppose the sets do not overlap).

In the instrumentation, the actions are prefixed with the identifier associated to the thread (see section 3.3.1). Join points associated to a non-skippable label e are instrumented in pseudo-code as follows, where `id` corresponds to the thread identifier:

```
1 monitorable.synchronizeOnEntry(id.e);
2 proceed();
3 monitorable.synchronizeOnExit(id.e);
```

Instrumentation can itself be seen as low-level aspects (this explains the pseudo-code `proceed()`). The variable `monitorable` corresponds to the monitorable object of the current base program thread, whereas the function `synchronizeOnEntry` is used to indicate to the monitorable object the intention of passing to a next state using the transition labeled *id.e*, and the function `synchronizeOnExit()` indicates the synchronization of on exit. The function `proceed()` indicates the execution of the original join point. Through this code, the monitorable object passes to the next state together with the other LTSs in the composition, using the transition labeled e.

Join points associated to a skippable label e are instrumented in pseudo-code as follows:

```
1  monitorable.synchronizeOnEntry(id.eventB_e);
2  monitorable.synchronizeOnExit();
3
4  monitorable.synchronizeOnEntry(id.proceedB_e, id.skipB_e);
5
6  if(monitorable.getChosen() == id.proceedB_e){
7    monitorable.synchronizeOnExit();
8    proceed();
9    monitorable.synchronizeOnEntry(id.proceedE_e);
10   monitorable.synchronizeOnExit();
11 }
12 else{
13   monitorable.synchronizeOnEntry(id.skipE_e);
14   monitorable.synchronizeOnExit();
15 }
16
17 monitorable.synchronizeOnEntry(id.eventE_e);
18 monitorable.synchronizeOnExit();
```

It is important to understand line 4. In this line, the monitorable object tries to pass to a next state using either `id.proceedB_e` or `id.skipB_e`. The transition chosen depends on the composition with other LTSs and the result is captured using `monitorable.getChosen()`. If the chosen transition is the beginning of a proceed then the original join point is executed (line 8), else the join point is skipped.

3.3.4 Operators

To introduce this section let us consider aspects as independent processes interpreting their LTS descriptions (as seen in Section 3.3.2). In the same way, let us also consider the base

program as an independent program describing its LTS. If all these processes run in an independent way, then they just become arbitrary interleaved processes. The objective of modeling them as LTSs is explained by applying the semantics of the LTS composition. Using this semantics, aspects are coordinated together with the base program, so that their actions are executed at the right time, following the parallel composition of their LTSs. However, a composition done only in this way (pure parallel composition) is sometimes unpredictable and may be not enough, *e.g.,* if they are two aspects which react differently to an event, one says *proceed* and the other *skip*, the result will randomly depend on which one first provides an answer. In order to solve this problem, operators are introduced, which provides a more predictable result.

A component approach has been chosen to model the application of operators. Two aspects can be combined using an operator, giving as a result a new aspect representing this combination. In this way, aspects are ordered in a hierarchy. The operators in this hierarchy impose extra-coordination by applying some LTS operations (such as relabeling and hiding events) over the aspects to which they have been applied.

In the compositional view, an aspect is either a *primitive aspect*, which constitutes the leaves of the hierarchy, or a *composite aspect*, which corresponds to a pair of aspects composed using some operator. In the implementation point of view, a primitive aspect is a process following its LTS, as explained in Section 3.3.2, whereas a composite aspect is a set of processes following the semantics of the composition of their LTSs.

3.3.4.1 An aspect as a component

An aspect has always an LTS describing its execution, which can be a single LTS or a composite one. This allows us to use the alphabet of this LTS as the interface of the component representing the aspect. Then for each skippable event e of the aspect, the component has the events *eventB_e*, *eventE_e*, *proceedB_e*, *proceedE_e*, *skipB_e* and *skipE_e* as interface. This describes the interface of an aspect no matter if it is a primitive or a composite aspect.

3.3.4.2 An aspect as a composite

The creation of a composite aspect is the result of applying an operator to a pair of aspects. What the operator does depends on the concrete semantic of the operator, however, it may consist of applying some LTS operations to the aspects. The important point is that the resulting composite aspect should expose an interface as described before.

3.3.4.3 The operator `ParAnd` as an example

Section 2.3.2 describes the operator `ParAnd`. It works by renaming `skip` and `proceed` messages. Both aspects share the events `eventB_e` and `eventE_e` so that the beginning and the end of advices are synchronized. Before (and after) `skip` or `proceed`, advices of the aspects are executed in parallel. The woven program is represented by the automaton of figure 2.10, where most of the synchronization events have been hidden after the parallel composition (except `eventB_update`, `proceedE_update` and `skipE_update`).

From a compositional point of view `ParAnd` is a composite aspect consisting of two component aspects. For each skippable action *e*, they define the events *eventB_e*, *eventE_e*, *proceedB_e*, *proceedE_e*, *skipB_e* and *skipE_e* as their interfaces. The aspect `ParAnd` renames the events in the interface of the component aspects and define a new LTS. Afterward it connects the renamed events of the aspect with the interface of the new LTS (seen as a component). Figure 3.4 illustrates the resulting composite aspect (for more clarity, it is not shown all the interface of the aspects). In the figure `C1` and `C2` are the component aspects and the resulting composite is the aspect `ParAnd`. The interface of both `C1` and `C2` has been renamed and the LTS `ParAndLTS` connected with them (the text in gray corresponds to the renamings).

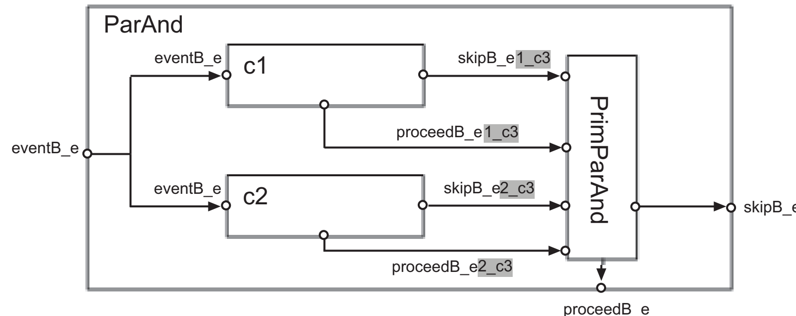


Figure 3.4: Example of synchronization of two components using `ParAnd`.

Our implementation considers composite aspects formed by two aspects. In this way, the following aspect in pseudo-code is permitted:
`Aspect = ParAnd(Aspect1,ParAnd(Aspect2,Aspect3))`
,the composition that results is shown in picture 3.5.

Possibly, supporting only two aspects in composite aspects could be a restriction. For example, expressing the previous aspect as `ParAnd(Aspect1,Aspect2,Aspect3)` could be more efficient. It is something that was not investigated in this thesis but could be interesting.

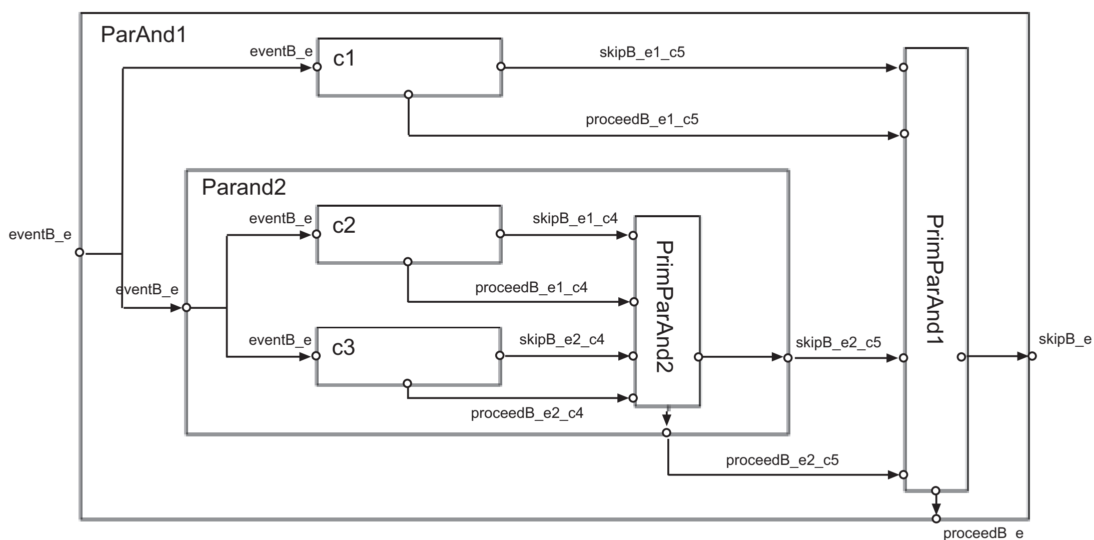


Figure 3.5: Example of synchronization of three components using `ParAnd`.

Chapter 4

A concrete implementation and optimizations

This chapter introduces a concrete implementation for CEAOP. The issues exposed in this chapter are the result of our experience with producing a running Java implementation of CEAOP.

As an intermediate layer, we have developed a Java library which makes it possible to associate a Java implementation to a finite state process. This implementation is presented in section 4.1.

Our concrete implementation translates both the Baton aspects and the Java base program into Java finite state processes which are composed using the previous referred Java library. This translation relies on Metaborg/SDF [16] to extend Java with Baton and Reflex [17] to instrument the base program.

We define three layers:

- In a first layer, Baton aspects and the Java base program are provided. Section 4.4 presents the assimilation of aspects, mappings and connectors defined in Baton.
- In a second layer (high-level layer), Baton aspects are represented as Java objects in a compositional way. A aspect hierarchy is created. This layer uses the poincuts defined by this hierarchy to intercede the base program using Reflex. This layer is detailed in section 4.2.
- In a third layer (low-level layer), aspects and base program are translated to objects implementing their LTSs. At this point, the intermediate layer with the Java library is used to perform the composition of the aspect and base program LTSs. This layer is detailed in section 4.3.

Finally, section 4.5 presents some optimizations.

4.1 A Java implementation of the LTS composition

This section exposes an implementation in Java of the composition of LTSs introduced in section 3.2. It explains how LTSs are implemented and details an implementation of LTS composition.

4.1.1 The monitor

The monitor is implemented as a class Monitor. It includes an instance variable selectedAction, which is non-null when the monitor is busy (its value is the selected action), and a collection of counters. It provides the following methods (which are public and synchronized):

- void register(PrimitiveLTS lts, List<Action> actions) is used to initialize the system. Before starting its own thread, each LTS, seen as an object implementing the abstract class PrimitiveLTS (see below), has to register to the monitor with this method. The parameter actions corresponds to the alphabet of the LTS. It is used by the monitor to set up the counters.
- void synchronizeOnEntry(List<Action> choice) is called by an LTS ready to perform a choice. It increments the appropriate counters. Whether the monitor is not busy and one of the counter bounds is reached, the monitor is set to busy. Then, the appropriate counters are decremented and the LTSs on the wait set of the selected counter are notified (their selectedAction instance variable is set to the selected action). For simplicity, this method is also used for the cases when there is a single action *i.e.,* not a real choice, then choices has only one element.
- void synchronizeOnExit() is called when an LTS has performed its part of the shared action. The counter corresponding to the selected action is decremented. When it reaches zero, the monitor is no longer busy. The LTSs on its wait set are notified. In a first implementation this method belongs to the class Monitor, however in an optimized version this method is separated in a class called Synchronized. This allows us to balance a little bit the charge of the monitor.

4.1.2 The LTS hierarchy

Each LTS extends the abstract class LTS, which includes the following methods:

- void start(), LTSs are supposed to be active objects. This method starts the LTS underlying thread.
- void relabel(String oldAction, String newAction), this method relabels an oldAction name by a newAction name. It is used by the operators (see below).
- void setMonitor(Monitor monitor), sets the monitor of the LTS and registers the LTS with the monitor. All LTSs are supposed to be synchronized by a monitor.

An LTS can be either a PrimitiveLTS or a CompositeLTS. The monitor of our implementation just works with PrimitiveLTSs, so that the abstract class PrimitiveLTS defines methods permitting the interaction with the monitor:

- void setAction(Action action), this method must be synchronized. It is used by the monitor to inform the PrimitiveLTS that it can proceed with the selected action action. It simply sets the selectedAction instance of the PrimitiveLTS and calls notifyAll to wake up the waiting LTS.
- Action getAction(), returns the selected action. If no action has been selected yet, the LTS thread executing the method waits in the LTS Java monitor until the notification of a new selection.
- void synchronizeOnEntry(List<Action> choice), initializes selectedAction to null, tells the monitor about the choice by calling the monitor version of synchronizeOnEntry, and finally waits for an action to be selected.
- void synchronizeOnExit(), simply calls the monitor version of synchronizeOnExit.

The prototypical implementation of PrimitiveLTS, the PrimitiveLTSImpl class, is a basic implementation of an LTS as an active object, *i.e.,* implementing the interface Runnable. It is based on a definition of the LTS by its alphabet (instance variable alphabet) and its transfer function, implemented as a combination of a hash map associating an action to an index (actionMap) and an array associating a (source) state and an index to a (target) state (target).

The implementation of setMonitor(Monitor) of PrimitiveLTSImpl sets and stores the monitor in an internal field. It also registers the LTS with the monitor. In addition, the implementation of start() starts the internal thread of a PrimitiveLTSImpl object. Once registered to the monitor and started, an instance of PrimitiveLTSImpl repeatedly evaluates the action choice associated to the current state, calls its method synchronizeOnEntry, calls its method void evaluate(int actionIndex), which, by

default, change states depending on the selected action, and calls its method `synchronizeOnEntry`.

The class `CompositeLTS` groups several `LTS` objects. Its implementation of `setMonitor(Monitor monitor)` *delegates* the method to its component LTSs, because the monitor can only work with primitive LTSs. In this way, the monitor is finally set in the leaves of the composition tree, which correspond to primitive LTSs. The implementations of `relabel(String, String)` and `start()` are also delegated to the methods of the component LTSs.

4.2 High-level layer: aspects and weaving

This section explains how aspects are represented in our concrete implementation of CEAOP. It also details how the weaving is configured and performed.

4.2.1 Implementing aspects

Concretely, an aspect consists of:

- An object that implements the LTS that models the aspect.
- The set of skippable actions of the aspect automaton.
- A map label-pointcut that permits to retrieve the pointcuts that are necessary for the instrumentation of the base program.

The LTS object and the actions are retrieved from the assimilation of a Baton aspect (see section 4.4.2). The map is retrieved from the assimilation of a Baton mapping (see section 4.4.3).

Aspects implement the interface `Aspect`, which defines the following methods:

- `Map<Action,Hookset> getPointcuts()`, returns the pointcuts used to instrument the base program. We use the class `Hookset` referring to the concept of pointcut in Reflex. This method returns an object that maps an action to a pointcut. Then the base program is instrumented for each action using its associated pointcut.
- `Set<String> getSkippable()`, returns the set of skippable actions in the automaton of the aspect. It is used to configure the instrumentation of the base program for an action, indicating whether the action is skippable or not. This method is also used in the low-level layer.

- `LTS getLTS()`, returns the object that implements the LTS representing the aspect. This object can be either an instance of `PrimitiveLTS` or an instance of `CompositeLTS`. This method is only used in the low-level layer.

The class `PrimitiveAspect` corresponds to the prototypical implementation of this interface. It represents a *primitive aspect*. It is instantiated giving a `PrimitiveLTS`, a list of skippable actions and the pointcuts as parameters. The assimilation of a Baton aspect generates subclasses of `PrimitiveAspect` (see section 4.4.2). This subclasses represent concrete aspects. In the assimilation of a Baton connector new instances of these classes are created(see section 4.4.4).

The definition of more complex aspects is done by using operators. An operator is represented as a composite aspect combining two or more aspects. It is implemented by subclassing `CompositeAspect`. In this way, a hierarchy of aspects is created such that its leaves correspond to instances of the class `PrimitiveAspect`. For instance, the operator ParAnd is implemented in the class `ParAnd`, which extends the class `CompositeAspect` and is instantiated by providing two `Aspect` objects and a mapping label-pointcut. The instantiation is done in the assimilation of a Baton connector.

The aspects in the high-level layer just permit the creation of one hierarchy of aspects. The real composition is done at the level of LTSs in the low-level layer.

4.2.2 Weaving using Reflex

In the preliminary work of this thesis, only one hierarchy of aspects is supported. This hierarchy is represented by a root `Aspect` object. This object understands the method `getPointcuts()` that returns the pointcuts used to instrument the base program.

We use Reflex [17] as the tool that allows this instrumentation using the pointcuts given by the aspect. Our necessities, with regard to instrumentation, are not complex, so that we could have chosen for instance AspectJ. We have chosen Reflex because it permits the configuration of the instrumentation using pure Java object. This makes the configuration of the instrumentation simpler for our purposes.

4.2.2.1 Overview of Reflex

Reflex started as a flexible model of *partial behavioral reflection* [18]. The clear connection between reflection and AOP led to the proposal of AOP kernels [19], which directed the posterior transformation of Reflex into a versatile kernel for multi-language AOP [17].

Reflex provides, in the context of Java, building blocks for facilitating the implementation of

different aspect-oriented languages so that it is easier to experiment with new AOP concepts and languages, and also possible to compose aspects written in different AOP languages. It is built around a flexible intermediate model, derived from reflection, of (point)cuts, links, and metaobjects, to be used as an intermediate target for the implementation of aspect-oriented languages. This is the level at which aspect interactions can be detected and resolved. Below this composition layer, a reflection layer implements the intermediate reflective model. Above the composition layer, a language layer, structured as a plugin architecture, helps bridge the gap between the aspect models and the intermediate model.

In order to be portable, Reflex is implemented as a Java class library. It relies on Javassist to weave hooks in the base bytecode at load-time and connect these hooks to the metalevel, or to add structural elements (methods, classes) according to a Reflex configuration program. Part of this configuration can be modified at runtime through a dynamic configuration API. Load-time configuration makes it possible to limit program transformation to the program points of interest (partial reflection with spatial selection). Runtime configuration makes it possible to activate/deactiveate the hooks (partial reflection with temporal selection), and access/change metaobjects.

An important property of Reflex is that the MOP of its underlying reflective layer is not fixed but can also be configured. This makes it possible to configure Reflex in order to support efficient static weaving but also makes it possible to support dynamic weaving (although a minimal overhead at the level of the hooks cannot be avoided after unweaving).

The primitive means to configure Reflex are configuration classes. To raise the level of abstraction, plugins can be provided: a plugin supports an aspect language, and is in charge of generating the appropriate Reflex configuration.

This thesis opt for using configuration classes due to the fact that the configuration needed is very simple.

Configuration classes The principle behind Reflection is the existence of a *base level* where the objects defined in the normal computation of the base program reside, and a *meta level* populated of *metaobjects*. Metaobjects permit the customization of the base level. The base program join points selected by a *cut language* delegate their executions to metaobjects, which perform some routine and resume the base program computation.

The cut language of Reflex is based on the definition of *hooks*. A hook is a point in the base program selected by a *class selector*, a *operation selector* and a *kind of operation*. A *hookset* permits grouping hooks in terms of common selectors. To clarify more the concepts, it is possible to do an analogy between join points and hooks, and pointcuts and hooksets.

Reflex allows us to intercede the base program by inserting `hooks`. It permits the passing of the execution control to metaobjects. The passing is in charge of *links*. A Reflex link

defines:

- A *hookset*, which indicates the places in the code where the control should be transfered to the meta level.
- A MOP (Meta Object Protocol), which indicates which metaobject should be called and how to call it.
- An activation condition, which permits to enable or disable a link at runtime.

The Reflex configuration is done in a class following a specific protocol, where hooksets, metaobjects and links are defined.

4.2.2.2 Reflex configuration of the weaving

From the hierarchy of aspects, it is possible to obtain a variable that maps actions to hooksets. These hooksets are defined in the assimilation of a Baton mapping. The configuration of Reflex uses these hooksets to configure the instrumentation of the base program.

To customize the base program, a metaobject is defined for each hookset. The metaobject is an instance of the class `ActionInstrumenter`, which is instantiated passing as parameter an action classified in skippable or non-skippable. The class `ActionInstrumenter` defines the method `instrument`, which corresponds to the instrumentation method.

The method `configureReflex` of a class named `ReflexConfigurer` implements the Reflex configuration for an aspect hierarchy as follows:

```
1  public void configure(Aspect aspect) {
2    Set<Action> skippable = aspect.getSkippable();
3    Map<Action,Hookset> hooksets = aspect.getPointcuts();
4    Iterator it = hookset.getKeySet().iterator();
5
6    while(it.hasNext()){
7      Action action = it.next();
8      Hookset hookset = hooksets.get(action);
9
10     ActionInstrumenter mo = new ActionInstrumenter(action, skippable.contains(action));
11
12     BLink link = API.links().createBLink(hs, new MODefinition.SharedMO(mo));
13     link.setControl(Control.AROUND);
14     link.setActivation(Activation.ENABLED_START_ON);
15     link.setCall(ActionInstrumenter.class.getName(),
16              "instrument",
17              new Parameter[] { Parameter.CLOSURE });
18
19     addBLink(link);
```

```
20    }
21  }
```

It is important to see the properties of the link. The control of the link is `Control.AROUND` (line 13), which implies that the join point interceded will be replaced by the execution of the metaobject method. This permits the use of `proceed`. The link starts activated (line 15). The parameter that the metaobject method receives is a Reflex closure representing the join point (lines 15-17). Some others parameters can be configured, for example to support the expression `target` of AspectJ.

4.3 Low-level layer: aspects and base program as LTSs

The low-level layer implements the modeling of aspects and base program as LTSs and their composition.

4.3.1 The LTSs modeling the aspects

An LTS exists associated to each aspect. The LTS associated to a `PrimitiveAspect` is an instance of a class named `RoutineLTS`, which extends `PrimitiveLTSImpl`.

The class `PrimitiveLTSImpl`, as explained in section 4.1.2, defines an implementation of LTS that interprets the LTS actions through the method `evaluate(int actionIndex)`, which evaluates the action having the index `actionIndex`. The default implementation of this method does nothing. The class `RoutineLTS` overrides the method `evaluate` in order to execute Java methods associated to actions. It defines a variable named `routines` that maps action labels to objects of the class `Method` of the reflection API of Java (this variable is set in the assimilation of a Baton mapping).

The method evaluate of class `RoutineLTS` is as follows:

```
1   public void evaluate(int actionIndex) {
2     Method method = routines.get(actionIndex);
3
4     if(method == null)
5       return;
6
7     Object[] params = ...;
8
9     try {
10      method.invoke(null, params);//execution of a static method
11    } catch (Exception e) {}
12  }
```

The routine to be executed is obtained (line 2) and executed (line 10). Line 7 corresponds to retrieving the parameters of the call. This is explained in section 4.4.5. In the preliminary work of this thesis, exception handling has not been implemented (line 11). It is important to note that not all actions have a routine associated, in such a case the evaluation does nothing (lines 4 and 5).

The LTS associated to a `CompositeAspect` is an instance of the class `CompositeLTS`. A `CompositeAspect` represents an operator between two aspects. It is instantiated passing the two `Aspect` objects as parameters. At instantiation time, it can apply the semantic of the operator by relabeling and/or hiding the actions of the `LTS` objects of each aspect and by optionally defining new `LTS` objects. The `CompositeLTS` of a `CompositeAspect` is formed of the `LTS` objects modified and the optional `LTS` objects defined.

For instance the creation of the `CompositeLTS` for the aspect `ParAnd` is as follows:

```
1  public class ParAnd extends CompositeAspect {
2    private Aspect aspect1;
3    private Aspect aspect2;
4    ...
5    public LTS getLTS() {
6      theLTS1 = aspect1.getLTS();
7      theLTS2 = aspect2.getLTS();
8
9      List<LTS> parAnds = new ArrayList<LTS>();
10     List<String> skipableList = getSkippable();
11     for(Iterator<String> it = skipableList.iterator();it.hasNext();) {
12       String skippable = it.next();
13
14       if(!hasSkippable(aspect1,skippable) || !hasSkippable(aspect2,skippable))
15             continue;
16
17       theLTS1.relabel("proceedB_"+skippable+id+"1","proceedB_"+skippable);
18       ...
19       theLTS2.relabel("proceedB_"+skippable+id+"2","proceedB_"+skippable);
20       ...
21
22       PrimitiveLTSImpl parAndLTS = ....;
23       parAnds.add(parAndLTS);
24     }
25
26     CompositeLTS composite = new CompositeLTS();
27     composite.add(theLTS1);
28     composite.add(theLTS2);
29     composite.addList(parAnds);
30     return composite;
31   }
32 }
```

The aspect `ParAnd` for each skippable action applies relabeling (lines 17-20) and creates a new `PrimitiveLTSImpl` (line 22). The LTSs created are collected in a list (line 23).

Afterward, the `CompositeLTS` is created that groups the renamed LTSs of the component aspects and the LTSs created by `ParAnd` (lines 26-29).

4.3.2 The LTSs modeling the base program

4.3.2.1 The class `Monitoreable`

The LTS representing a base program thread is implemented using the class `Monitorable`, which extends the class `PrimitiveLTS` and implements the inherited methods `start` and `relabel` as empty methods. The class `Monitorable` is different than the class `PrimitiveLTSImpl` because the latter synchronizes and interprets the transitions of a predefined automaton. However, the `Monitoreable` class just synchronizes in the transitions that the instrumented base program indicates.

Each thread in the base program has its own instance of the class `Monitoreable`. This is possible by using thread locality in Java. The `ThreadLocal` variable `Monitorable.THREAD_LOCAL` stores the object `Monitorable` associated to the corresponding thread. The constructor of an object `Monitorable` receives as parameter an identifier which is obtained using other thread local variable, `Monitorable.ID`, which returns an unique identifier for the thread.

The instantiation of the object `Monitorable` of a thread could be done in two ways. The first one would consist in instrumenting some routine of the base program and do the instantiation the first time a thread executes such a routine, *e.g.,* the method `run()` of a class implementing `Runnable`. It implies to add some expression at the language level in order to indicate a routine to be instrumented and the alphabet of the objects `Monitorable` to be instantiated for the threads executing such a routine. A second option is simpler. It consists in instantiating the object `Monitorable` in the instrumentation code of the first instrumented action that the thread executes, *e.g.,* if the action `login` has been instrumented, then the instantiation is performed the first time a client executes a join point belonging to the pointcut associated to `login`.

We opt for the second alternative. However, in this case the alphabet for the object `Monitorable` is not known. Only one action of the alphabet, which corresponds to the instrumented action, is known. To circumvent this, each time the thread executes an action that has not been included in the language of the object `Monitorable`, the action is included in its alphabet and the object is subscribed to the action with the monitor. It is done using the method `register(PrimitiveLTS,List<Action>)` of the monitor.

The code below corresponds to the code that instantiates the object `Monitoreable` of a thread:

```
Action action = ...;
Monitor monitor = ...;
List<Action> choice = new ArrayList<Action>();
choice.add(action);
Monitorable monitorable = Monitorable.THREAD_LOCAL.get();

if(monitorable == null) {
  Integer ID = Monitorable.ID.get();
  monitorable = new Monitorable(ID);
  monitorable.addAction(action);
  Monitorable.THREAD_LOCAL.set(monitorable);
  monitor.register(monitorable,choice);
  monitor.synchronizeOnEntry(choice, monitorable);
}
```

Lines 1 and 2 correspond to the declaration of the action and the monitor, respectively, which is not detailed in this section. Line 5 corresponds to retrieving the object `Monitorable` of the current thread using the thread local variable `Monitorable.THREAD_LOCAL`. If the object has not already registered in the thread local variable, the result is null and it has to be instantiated and registered. Line 8 shows the definition of a new identifier for the object, which is given as a parameter in its instantiation (line 9). Lines 10 and 11 include the action in the alphabet of the object and the object is registered in the thread local variable, respectively. Finally, line 12 and 13 perform the registering and the synchronization on entry with the monitor, respectively.

4.3.2.2 Why using the class `Monitorable`

The advantages of using our schema (with an object `Monitorable` for each thread) is related with the advantages of using thread locality. As an invariant of any possible schema, each thread has an object implementing its LTS (a `PrimitiveLTS` object). When a join point is interceded in the base program, such an intercession is done in the context of some thread (but not necessarily in a method of the class implementing `Runnable`). Then, we need to look for the `PrimitiveLTS` of such a thread. Thread locality becomes the unique possible way to easily accede to the `PrimitiveLTS`. Since the activities of the `PrimitiveLTS` objects associated to the threads are the same no matter the thread, we have opted for sub-classing `PrimitiveLTS` with a standard implementation, the class `Monitorable`.

For example suppose the following code:

```
public class Client implements Runnable {
  private Brower b;
  ...
  public void run() {
    this.login();
```

```
6       b.browse();
7     }
8
9     public void login() {
10      ...
11    }
12  }
```

The code above represents a client as an active object. It can perform the actions `login` and delegate the action `browse` to an object of class `Browser`.

A more static approach could transform the class `Client` into a `PrimitiveLTS`, for instance, by performing both structural and behavioral transformations using Reflex. Then in the instrumentation of the execution of a method such as `login`, it could be possible to assume that the target object (in this case the object `Client`) corresponds to the `PrimitiveLTS` of the thread (if clients only log in in their own threads). However, if we want to instrument the execution of the method `browse`, an easy way to obtain the `PrimitiveLTS` of the thread is by using thread locality. Using thread locality we could retrieve the `PrimitiveLTS`, which, in this case, would be the object `Client`. Our approach, instead of transforming a class like `Client`, uses a predefined class `Monitorable`.

This is related to why for each thread there is an object `PrimitiveLTS` associated. The reason can be explained by negation. If two threads share the same object `PrimitiveLTS`, then both threads could concurrently notify an action. Since the `PrimitiveLTS` implements a single process in LTSA, it is by definition sequential. Therefore, the actions notified by two distinct threads should be sequentialized. It could be undesirable. By implementing a `PrimitiveLTS` for each thread more concurrency is allowed.

4.3.2.3 Code of the instrumentation

The section 4.2.2 explained that the instrumentation of the base program is done by interceding join points in the base program, using Reflex, and passing the control to a metaobject of class `ActionInstrumenter`. The method that receives the control is the method `instrument`. This method replaces, as an around, the execution of the join point. In addition, this join point has been associated to an action (that is the action associated to the metaobject in the variable `action`) and a boolean indicating if the action is skippable or not (the boolean corresponds to the variable `skippable` of the metaobject). How the join point is interceded depends on the kind of action, as introduced in section 3.3.3. In order to illustrate how an action is instrumented in Reflex, we show the code of the method `instrument` as follows:

```
1  public Object instrument(IExecutionPointClosure aClosure)
2  {
```

```
3    Object result = null;
4    Monitoreable lts = Monitoreable.get();
5    Integer id = lts.getID();
6    String action = this.action;
7
8    if(!this.isSkippable()) {
9      lts.synchronizeOnEntry(id + "." + action);
10     result = aClosure.proceed();
11     lts.synchronizeOnExit();
12   }
13   else
14   {
15     lts.synchronizeOnEntry(id + "eventB_" + action);
16     lts.synchronizeOnExit();
17
18     lts.synchronizeOnEntry(id + "proceedB_" + action, id + "skipB_" + action);
19
20     if (lts.getChosen().getName().equals(id + "proceedB_" + action)) {
21       lts.synchronizeOnExit();
22
23       result = aClosure.proceed();
24
25       lts.synchronizeOnEntry(id + "proceedE_" + action);
26       lts.synchronizeOnExit();
27     }
28     else {
29       lts.synchronizeOnExit();
30
31       lts.synchronizeOnEntry(id + "skipE_" + action);
32       lts.synchronizeOnExit();
33     }
34
35     lts.synchronizeOnEntry(id + "eventE_" + action);
36     lts.synchronizeOnExit();
37   }
38
39   return result;
40 }
```

Line 5 obtains the identifier used to prefix all the actions (see section 3.3.1). Line 4 is used to obtain the object `Monitoreable` of the current thread. This object is used to perform the synchronizations. If the action of the metaobject is not skippable, then it is only required a simple synchronization and a proceed (lines 9-11). If the action is skippable then the action is split in several synchronization events (lines 15-36).

4.3.3 Configuring the composition

In a low-level the composition of aspects and its application to the base program is translated to the composition of the LTSs modeling the aspects and the LTSs modeling the base program. The composition is done using the implementation of our library for LTS

composition. Therefore, the composition consists in the instantiation of a monitor (one for each hierarchy of aspects) that has to be shared by the aspect LTSs and the base program LTSs. The latter is done in two steps. First, the monitor is given to the aspect LTSs straight after an aspect hierarchy has been instantiated. In a second step, each time a new `Monitoreable` object is instantiated, such an object is incorporated to the composition.

The work of this thesis considers only one hierarchy of aspects, which is represented by a root `Aspect` object (that can be a composite aspect). From this object, the corresponding object `LTS` (that can be a composite LTS) is possible to retrieve. This is done using the method `getLTS()`. The composition is configured using a `Monitor` object for the hierarchy of aspects and by setting the object as the monitor of the aspect LTS using the method `setMonitor(Monitor)`. Afterward, by invoking the method `start()` over the `LTS` object, the different LTSs composing it begin to run and synchronize with the monitor.

The following code configures the composition for a given aspect `aspect` in the assimilation of a Baton connector:

```
1 Monitor monitor = new Monitor()
2 LTS lts = aspect.getLTS();
3 lts.setMonitor(monitor);
4 lts.start();
```

The variable `aspect` represents the root of a hierarchy of aspects. Line 1 instantiates a monitor for the hierarchy. The LTS of the hierarchy is obtained (line 2). The monitor is set (line 3). The LTS is started (line 4).

4.4 Implementing the DSAL

This section describes the implementation of Baton, introduced in section 3.1.3. The implementation is done using Metaborg, a methodology to extend language syntax, which in our case is used to extend Java. In the remainder, Metaborg is presented and afterward we detail the result of the assimilation of each part of the Baton language.

4.4.1 Overview of Metaborg

Metaborg [16] is a method for providing concrete syntax for domain abstractions to application programmers. The method consists of embedding domain-specific languages in a general purpose host language and assimilating the embedded domain code into the surrounding host code. Instead of extending the implementation of the host language, the

assimilation phase implements domain abstractions in terms of existing APIs leaving the host language undisturbed. Indeed, MetaBorg can be considered a method for promoting APIs to the language level.

MetaBorg consists of SDF [20] (Syntax Definition Formalism) for modular syntax definitions, and Stratego/XT [21] for language assimilation. A feature of SDF is that it inverts productions in the syntax definition: instead of having a single production with alternatives per non-terminal (separated with `|` in BNF), SDF definitions have one production per alternative. This makes it possible for an SDF module to introduce new alternatives for a given non-terminal without the need to modify the original definition, hence fostering syntactic extensibility of the language.

The process of assimilation corresponds to the translation of syntactic extensions to the general-purpose code. The next sections details the result of the process of assimilation for each part of Baton.

4.4.2 Aspects

The assimilation of the automaton defined in a Baton aspect is done using an internal representation of FSP, through classes such as `FSP`, `FSPProcess`, `FSPSequence`, etc. which are not detailed in this report. The assimilation is performed through the following actions:

- The proper waiting loops are introduced.
- The transitions are prefixed, as explained in section 3.3.1.
- The skippable actions are split up in the synchronization events, as explained in section 2.3.
- The resulting FSP representation is translated into a concrete LTS.

The process of assimilation allows us to obtain an instance of the class `PrimitiveLTSImpl` and a list of skippable actions. The process of assimilation also generates a subclass of `PrimitiveAspect`, that uses the `PrimitiveLTSImpl` instance and the list of skippable actions as templates to generate new aspect instances. For instance, for the example of the e-commerce application of section 2.3 the subclass for the aspect `Consistency` is as follows:

```
1 public class Consistency extends PrimitiveAspect {
2   private static PrimitiveLTSImpl ltsTemplate;
3   private static List<Action> skippableTemplate;
4
```

```
5    public Consistency(Map<Action,Method> routines, Map<Action,Hookset> pointcuts) {
6      super(new RoutineLTS(ltsTemplate, routines), skippableTemplate, pointcuts);
7    }
8  }
```

Lines 2 and 3 define the static variables used as templates to create new instances of the aspect `Consistency`. Line 6 invokes the constructor of the class `PrimitiveAspect` by passing a new instance of `RoutineLTS`, the skippable actions and the pointcuts.

The assimilation of a parametrized aspect, such as the one shown in the code below, is done by generating all possible combinations for the variables defined by the process. In each combination, the expression `when` indicates if, for the corresponding process, a sequence is available or not.

```
1  aspect Consistency2 {
2
3    p[logged:0..1][updated:0..1] =
4       when (logged == 0 && updated == 0)
5         update -> p[logged][1]
6       | when (logged == 1 && updated == 0)
7           update(admin) > skip; log(admin) -> p[logged][1]
8       | login -> p[1][updated]
9       | checkout -> p[0][updated]
10 }
```

The code declares a distinct version of the aspect `Consistency`, such that only one update is allowed. This is done using a variable that indicates whether an update has been performed or not.

The assimilation of the previous aspect is analogous to the assimilation of the following aspect:

```
1  aspect Consistency2 {
2
3    p00 =
4       update -> p01
5       | login -> p10
6       | checkout -> p00,
7
8    p10 =
9       update(admin) > skip; log(admin) -> p11
10      | login -> p10
11      | checkout -> p00,
12
13   p01 =
14      | login -> p11
15      | checkout -> p01,
```

```
16
17   p11 =
18       | login -> p11
19       | checkout -> p01
20 }
```

4.4.3 Mappings

4.4.3.1 Maps label-pointcut and label-function

The assimilation of the pointcuts of a Baton mapping generates part of the code used to instrument the base program. This code implements the creation of hooksets that declare the right class and operation selectors.

The assimilation of the functions defined on a mapping produces a map label-function that is used afterward in the instantiation of an aspect.

A class with the name of the mapping is created. This class declares both the method `getMapLabelFunction()` returning an object mapping a label to a Java method, and the method `getMapLabelPointcut()` returning an object mapping a label to a Reflex hookset.

4.4.3.2 Generation of hooksets

In Reflex the cut language is implemented defining *hooksets*, which consist of a *class selectors* to select classes, a *operation selector* to select operations, and the kind of operation to be selected such as `MsgReceive`, `MsgSend`, among others. The code is instrumented in such a way that when the program execution reaches selected join points, the control flow is passed to a *metaobject*, which executes some routine and then resumes the execution of the base program.

For the AspectJ expressions `execution` and `call`, the operations forming the hooksets are `MsgReceive` and `MsgSend`, respectively.

The expressions nested inside these expressions consist of a modifier, a return type, a class type, a member name and some class types describing parameters. For these elements, distinct selectors are defined. The operation selectors `PublicOS` and `PrivateOS` select class members such that their modifiers are `public` and `private`, respectively. The operation selector `ReturnOS` permits to select the members by their return type. The operation selector `NameOS` selects the members by their name. Finally, the class selector `NameCS` selects the classes by its name.

For a given expression, an operation selector `AndOS` is instantiated. This selector receives

as parameter the corresponding operation selectors and performs the inclusion of their selections.

In the case of the return type, the member name and/or the parameters are declared using a wildcard, the corresponding operation selectors are omitted. If all operation selectors are omitted then the operation selector `AnyOS`, selecting any member, is used. In the same way, when the class type is declared with the wildcard the class selector `AnyCS`, selecting any class, is used.

For example, the Reflex hookset for an action `update` mapping the pointcut `execution(public void Admnistrator.update(..))` is as follows:

```
1 PrimitiveHookset hs = new PrimitiveHookset(MsgReceive.class,
2             new NameCS("Administrator"),
3             new AndOS(new OperationSelector[] { new PublicOS(),
4                                                 new ReturnOS(ReturnOS.VOID),
5                                                 new NameOS("update") }));
```

4.4.4 Connectors

The assimilation of Baton connectors consists in the creation of aspect instances and the configuration of the base program instrumentation. An aspect instantiation means a relationship between a Baton aspect and a Baton mapping. This relationship permits the following:

- The instantiation of aspects by providing them with the routines (map label-function) defined in the mapping.
- The creation of an aspect hierarchy represented by a root aspect.
- To provide the root of the aspect hierarchy with the pointcuts (map label-pointcut) defined in the mapping.
- The configuration of Reflex, using the method `configure` of the class `ReflexConfigurer`.

For example, if a connector declares a new aspect instance using the syntax:

```
Aspect a = new ParAnd(new Consistency(), new Safety()) using MyMapping;
```

the code of the assimilation of the mapping is as follows:

```
1 Map<Action,Method> routines = MyMapping.getMapLabelFunction();
2 Map<Action,Hookset> hooksets = MyMapping.getMapLabelPointcut();
3
4 Aspect consistency = new Consistency(routines,null);
5 Aspect safety = new Safety(routines,null);
6 Aspect parAnd = new ParAnd(consistency, safety, hooksets);
7
8 ReflexConfigurer.configure(parAnd);
```

This code requires the previous assimilation of the aspect and the mapping files, so that the classes `Consistency` and `MyMapping` have been generated. In line 1 the routines required by the instantiation of the aspect are retrieved. This is done by calling the method `getMapLabelFunction()` of the mapping. In line 2 the pointcuts required by the instantiation of the root of the hierarchy of aspects are retrieved. This is done by calling the method `getMapLabelPointcut()` of the mapping. Lines 4 to 6 define the hierarchy of aspects. The root of the hierarchy is the aspect `parAnd`. The primitive aspects `consistency` and `safety` do not require the pointcuts since they are not roots in the hierarchy the aspects. Finally, line 8 invokes the method that configures Reflex passing the root as parameter (see section 4.2.2).

4.4.5 Implementing aspect binding

The DSAL allows us to define aspects such as:

```
aspect UpdateLogger {
  p = update(admin, db) > proceed; log(db) -> p
}
```

This aspect allows us to log all the updates performed in a database `db` by an administrator `admin`. The action `update` is extended with an action declaring the after advice `log`. The parameter `db` of `update` is passed to the action `log` as its first parameter.

The implementation of this feature requires the binding of the parameters defined by the action `update`. This binding is done in the instrumentation code generated by the assimilation of the pointcut associated to `update` in a mapping. Then, the execution of the routine associated to the advice `log` requires to receive as parameter the value of the binding of the second parameter of `update`.

We define a class called `Closure` in order to implement this feature. When the pointcut associated to a label that defines parameters is assimilated in a mapping, a new instance of the class `Closure` is created. This object is mapped to the label of the pointcut and

stored in the variable `closures` of the class generated for the mapping. The class `Closure` stores the values of the variables bound in the pointcut.

On the other hand, in the assimilation of the aspect, for each advice action declaring parameters, an array with the indexes of the parameters received by the action is created. This array is stored in the variable `argIndexes` of the class generated for the aspect.

The evaluation of an advice action in the method `evaluate` of the class of the aspect looks for the closure associated to the corresponding pointcut in order to obtain the value of the bound variables. Using the array of indexes associated to the advice action it is possible to obtain the values of the corresponding parameters.

For the example, the method `evaluate` would be the following:

```
1  public void evaluate(int actionIndex) {
2    Method method = routines.get(actionIndex);
3
4    if(method == null)
5       return;
6
7    Action action = actions.get(actionIndex);
8    Closure closure = mapping.getClosure(action);
9    int[] argIndexes = this.argIndexes[currentState][actionIndex];
10   Object[] params = closure.getArgs(argIndexes);
11
12   try {
13     method.invoke(null, params);//execution of a staticmethod
14   } catch (Exception e) {}
15 }
```

4.5 Optimizations

4.5.1 Avoiding waiting loops in aspects

The model introduced in CEAOP uses waiting loops in the definition of the LTSs that describes aspects. The waiting loops are used in order to avoid deadlocks. The introduction of waiting loops, in a declarative way, is as follows: let us consider P, the pseudo-LTS of an aspect (without waiting loops), and its alphabet αP. For each state s of P, let $actions(s)$ be a function providing us with the transitions for the state s. For each action $a \in \alpha P$ such that $\forall t \in actions(s)$, t does not transit using a, a waiting loop, corresponding to the transition (s, a, s), is added. The result is P', the proper LTS describing the aspect(with waiting loops). In the remainder of this section we refer to an LTS in the sense of a proper LTS describing an aspect.

A waiting loop allows us to ensure that each state of an LTS define a transition for each action in the alphabet. Its function is to synchronize with the other LTSs in actions that are not relevant for the LTS in its current state, avoiding deadlocks. However, it does not have any effect on the LTS: a waiting loop does not introduce any routine and it does not imply any change of state.

However, waiting loops do have an effect in the interpretation of LTSs. They introduce extra synchronization in transitions that are not relevant for the interpretation of an LTS, making such an interpretation slower.

A clear optimization is to make an LTS to avoid the synchronization on waiting loops. Or even better, to avoid the use of them.

The solution proposed consists in a modification of the implementation of section 3.2 of LTS and its composition, in order, to avoid the use of waiting loops. This solution is done around the following observation supposing the absence of waiting loops. When an LTS P_1 tries to transit to a next state using an action a, it has to be synchronized with all the other LTSs having a in their alphabets. If an LTS P_2 has a in its alphabet but it does not define any transition on a in its current state, then P_1 has to wait until P_2 reaches an state where a transition on a is defined. If the arrival to such an state depends on the transition of P_2 to another state, it never occurs and produces a deadlock. If when P_1 tries to transit on a, P_2 does not declare the action a in its alphabet, then P_1 does not require a synchronization with P_2.

The solution consists in a schema where LTSs declare a subset of their real alphabet. These subsets are changed by them in the transition to next states. This solution is such that the alphabet of an LTS in a given state contains only the actions defining transition in such an state. In this way, instead of synchronizing in a waiting loop producing any affect, the LTS hides the action so that the other LTS do not synchronize with it.

In terms of our concrete implementation, the subscription and unsubscription to action is performed in the method `synchronizeOnExit(List, List)` of the class `Monitor` which has been modified by providing it with two lists of actions as parameters. Each time an LTS changes its state and tries to synchronize on the exit of a transition, it calls `synchronizeOnExit(unsubscribe, subscribe)`, which provides the monitor with the list `unsubscribe` of the actions corresponding to its previous choice, and the list `subscribe` of the actions corresponding to the transitions forming the choice of the new state. The monitor takes the actions of the first list and the actions of the second list, and decrements and increment their bounds, respectively. When for the new state there is a single internal action which is not visible to the other LTSs (typically an advice), the LTS include in the `subscribe` list the actions of the next state that defines some visible action.

The use of the method `synchronizeOnExit` avoids the inclusion of a new synchronized method to subscribe and unsubscribe to actions. Furthermore, we take advantage of the

fact that when this method is called in the respective LTS, the corresponding synchronizations on entry have already been performed by all the LTSs and the monitor is busy, so that the bounds are not being used.

4.5.2 Optimizing the object `Monitorable`

Section 4.3.2.1 introduced the object `Monitorable` implementing the LTS that represents a base program thread. We have chosen a schema such that each object `Monitorable` registers the actions of its alphabet in a *lazy* way, *i.e.,* when an instrumented action is reached by the thread. This is because the complete alphabet of the `Monitorable` cannot be known a priori. Each time a new action is known for the LTS alphabet, it is invoked `register(PrimitiveLTS,List<Action>)` on the monitor to make the LTS subscribe to the action.

In order to avoid the use of the method `register(PrimitiveLTS,List<Action>)` defined in the class `Monitor` to subscribe to actions, which implies an extra point of synchronization in the monitor, we create a new instance of the method `synchronizeOnEntry` by providing it with a new parameter that corresponds to the object `Monitorable`. In this way, this method is used when the object synchronizes on entry with actions that the monitor has not associated to this object. This method first subscribes the object to the actions that the monitor has not associated to this object yet and then performs the synchronization.

4.5.3 Eliminating the double synchronization

Section 3.2.5 explained the necessity of a double synchronization schema for actions. In the implementation, this is done through the methods `synchronizeOnEntry` and `synchronizeOnExit`. The rol of the latter is making the LTSs participating of a synchronization meet at the end of the transition. This is for ensuring that all LTSs have transited together.

In some cases the synchronization on the exit on a shared action can be eliminated. This is the case when in the next state of the corresponding transition, all the LTSs share the same actions. Therefore, they have necessarily to meet again in the synchronize on the entry of a next action.

Chapter 5

The Readers and Writers problem

As an illustration, this chapter exposes a solution of the classical problem of Readers and Writers using CEAOP.

5.1 The problem

The Readers-Writers problem is concerned with access to a shared data by two kind of processes. Readers execute atomic actions that examine the data while writers both examine and update the data. For the shared data to be updated correctly, writers must have exclusive access to the data while they are updating it. If no writer is accessing the data, any number of readers may concurrently access it.

5.2 A solution using CEAOP

The solution to this problem, exposed in this section, is based on the solution proposed by [7], which is modeled using LTSA and then implemented "by hand". However, our aim is to generate a more transparent solution by using AOP, which is transparent to the programmers of the base system. The programming of readers and writers is simplified if concerns about access to the shared data are omitted. Afterward, the use of aspects allows us to use these concerns in a more modular and straightforward way.

Our solution is based on the intercession, using aspects, of the operations of reading and writing in the shared data. The action `read` is split up in the acquisition of a lock for reading, the execution of the read and the release of the lock. Something similar is done for the operation `write`.

The aspect interceding `read` is as follows:

```
1  aspect Reader {
2    p = read > +acquireRead; proceed; +releaseRead -> p;
3  }
```

The actions `acquireRead` and `releaseRead` are prefixed by the sign `+`, which indicates that this actions are visible to the other aspects in the composition. The aspect interceding `write` is analogous:

```
1  aspect Writer {
2    p = write > +requestWrite; proceed; +releaseWrite -> p;
3  }
```

In order to coordinate the exclusive access of writers to the shared data, an aspect is implemented. This aspect, composed with the previous aspects, orders the reads and writings to the data. It keeps a count of the number of readers accessing the data and whether a Writer is writing. The action `acquireRead` is only available in a state such that no Writer is accessing the data, in the same way the `acquireWrite` is available in states such that no Reader is reading. The aspect implementing the lock is the following:

```
1  aspect Lock {
2
3    rw[readers:0..10][writing:0..1] =
4      when (writting == 0)
5        acquireRead -> rw[readers + 1][writing]
6      |releaseRead -> rw[readers - 1][writing]
7      |when (readers == 0 && writing == 0)
8        acquireWrite -> rw[readers][1]
9      |releaseWrite -> rw[readers][0].
10
11 }
```

The advantage with regard to a pure OO solution (as used in [7]) or an AspectJ is the fact that it relies on a high-level DSAL, close to a specification language, and therefore less error-prone. It is also simpler to reuse.

Chapter 6

Conclusions

6.1 Contributions

The main contribution of the work developed in this thesis corresponds to the preliminary implementation of CEAOP, in the form of a small DSAL. This implementation has allowed a deeper analysis of the concept formulated by the approach.

One of the products of this thesis corresponds to the development of a library that permits the translation of FSP processes and their composition into running Java processes, composed following the FSP semantics. Each process is implemented as an active object interpreting the actions of its LTS. A centralized monitor is in charge of the composition. This implementation is simple and correct since it strictly follows the semantics of LTS composition.

6.2 Perspectives

6.2.1 Improvements to the implementation

Improving concurrency

The LTS composition serializes the composition in a single LTS representing the synchronous product. Since we take care of following its transitions in a deterministic way, the monitor may only take one decision at each time. So that, just one transition is chosen. The result is that only one shared action is concurrently performed.

Our LTS implementation needs concurrently executing different shared actions. More parallelism would imply to reformulate the invariants of the implementation of the monitor, for example, when the monitor is busy because some decision has been taken, it would still be possible to choose another action.

However, it presents the problem of being too much centralized, which is a drawback in the case of distributed systems. For instance, for our example of the e-commerce application, clients and administrators can be implemented as active LTSs. If the monitor resides in a distinct host, it is mandatory a remote communication each time these actors need to transit in their automata.

Intersection of pointcuts

The DSAL implemented in this thesis separates the automaton definition and the pointcut definition of aspects. The main reason for that is the composition of aspects following the same semantics for labels. Then if two aspects define a shared action, there is an only one pointcut for such an action. However, this does not impede the non-empty intersection of the sets of join points. Our implementation assumes that the set of join point of a pointcut does not intersect with the set of join points of another pointcut. However, if a given join point is matched by two pointcuts, the resolution of the conflict is up to the default composition strategy of the instrumentation tool used (in our case Reflex). In some case one action is synchronized first and after the other.

For example, suppose that an action `logAll` defines a pointcut `call(* *.*(..))` and an action `login` defines a pointcut `call(* *.log(..))`. When a method `Client.log()` is invoked in the context of some base program thread, both actions are valid. In our current implementation one action is synchronized first and after the other. However, the LTS representing the thread should notify to the monitor a choice consisting of both actions. The implementation of this feature impedes instrumenting the base program independently for each pointcut. It is necessary a coordination between pointcuts to determine if there is an interaction. The feasibility of the implementation depends on the expressiveness of the tool used to instrument the base program.

Extension to the Baton language

Our Baton language allows us to define the automaton of an aspect in a Baton aspect. The syntax permits define FSP processes such as the following:

```
safe = minimize(editor) > save(editor);proceed -> safe.
```

This process can be used to model an aspect that each time a certain text editor is minimized makes the editor save. Afterward, in a Baton mapping the action label save can be associated to certain Java method that receives the object representing the editor and sends the method save to the object. In the actual implementation only static methods are supported. An interesting approach is to permit something more Object-oriented such as:

```
safe = minimize(editor) > editor.save;proceed -> safe.
```

Then a parameter could be used as a call target.

General remarks

- The prototype does not implement any protocol for handling exceptions. A typical place where exception handling is needed, corresponds to the execution of the advices routines in the implementation of an aspect LTS. The routines are defined in a Baton mapping. In the assimilation of the mapping could be good to check the existence of the defined Java methods. Afterward, the failed execution of a routine should be handled following some defined criteria, making the prototype more robust.
- The prototype has, for the time being, only be used on very few examples, such as on our model of e-commerce application and on a solution of the problem of Readers and Writers. It is necessary to use it on more examples, of different sizes, in order to better assess it. In turn, these new examples will probably give new ideas on how to improve it.
- Section 4.5 proposes some optimizations to the prototype, specially with respect to the library implementing LTS composition. A future work is to benchmark the effectiveness of the approaches.

6.2.2 Improvements to the CEAOP model

Advice expressiveness of aspects in CEAOP

The basic language considered by CEAOP is limited. It would be interesting to consider extensions to this basic language. For instance, we have seen that an advice takes the form *before_action proceed_or_skip after_action*, *i.e.,* it has to explicitly indicate either the keyword `skip` or `proceed`. This keyword is fixed for each advice, as a result, an aspect cannot dynamically decide whether it should skip or not its corresponding join point executions. More dynamic advices (such as the `around` of AspectJ), which make the `proceed` conditional on runtime context, cannot be modeled using this approach. This

advice restriction makes the LTS representing an aspect impose a fixed transition to the LTS modeling the base program. To wit: whereas the latter defines a choice between skip or proceed, the former takes a fixed decision by defining a single transition on either skip or proceed. Therefore, the decision is taken at the LTS level.

Pointcut expressiveness of aspects in CEAOP

CEAOP models aspects and base program using transition labels that denote events in the base program. No much more it is said about how such transition labels are mapped to join points in the base program. Our implementation associates such transition labels to pointcuts that matches just base program syntax. However, the necessity of support for dynamic pointcuts has been early stated in the AOP state of the art [22, 23]. A support of dynamic pointcuts in CEAOP necessary implies to review some semantic details.

Bibliography

[1] Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, September 2004.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241, Jyväskylä, Finland, June 1997. Springer-Verlag.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. Number 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[4] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. Research Report RR-5873, INRIA, March 2006.

[5] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Towards a model of concurrent AOP. SPLAT'06, March 2006.

[6] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 141–150, New York, NY, USA, March 2004. ACM Press.

[7] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley&Sons, 1999.

[8] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[9] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel*

Architectures and Separation of Crosscutting Concerns, pages 170–186, Kyoto, Japan, September 2001. Springer-Verlag.

[10] Wim Vanderperren, Davy Suvée, María Agustina Cibrán, and Bruno De Fraine. Stateful aspects in jasco. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2005.

[11] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming Systems Languages And Applications*, pages 345–364, San Diego, CA, USA, October 2005. ACM Press.

[12] 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, Newport Beach, CA, USA, October 2004. ACM Press.

[13] Simplice Djoko Djoko, Rémi Douence, Pascal Fradet, and Didier Le Botlan. CASB: Common Aspect Semantics Base. Technical Report AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France, February 2006.

[14] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, Chicago, Illinois, USA, March 2005. ACM Press.

[15] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer-Verlag, 2005.

[16] Martin Bravenboer, René de Groot, and Eelco Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Braga, Portugal, July 2005.

[17] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, Lecture Notes in Computer Science, Tallin, Estonia, September 2005. Springer-Verlag. To appear.

[18] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L.

Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[19] Éric Tanter and Jacques Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, September 2004.

[20] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Amsterdam, 1997.

[21] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[22] J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. In *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)*, June 2000.

[23] B. De Fraine, W. Vanderperren, D. Suvee, and J Brichau. Jumping aspects revisited. In *Proceedings of DAW 2005*, March 2005.