Aspectualizing Component Models : implementation and Interferences Analysis

Abdelhakim Hannousse

l'université nantes angers le mans

PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

ECOLE DES MINES DE NANTES

THESE

Abdelhakim Hannousse

ECOLE DOCTORALE : ED STIM
THESE N° 2011EMNA0009

Thèse présentée en vue de l'obtention du grade de
Docteur de l'Ecole des Mines
Sous le label de l'Université Nantes Angers Le Mans
Discipline ***Informatique***

Soutenue le 14 novembre 2011

Aspectualizing Component Models : Implementation and Interferences Analysis

DIRECTEUR DE THESE **:**
Mario Südholt, Professeur, EMN
CO DIRECTEUR DE THESE :
Rémi Douence, Maître de conférence, INRIA, EMN
Gilles Ardourel, Maître de conférence, Université de Nantes

RAPPORTEURS DE THESE :
Laurence Duchien, Professeur, Université de Lile 1 (USTL)
Isabelle Borne, Professeur, Université de Bretagne Sud

PRESIDENT DU JURY :
Jean-Marc Jézéquel, Professeur, Université de Rennes

MEMBRES DU JURY :
Laurence Duchien, Professeur, Université de Lile 1 (USTL)
Isabelle Borne, Professeur, Université de Bretagne Sud
Mario Südholt, Professeur, EMN
Rémi Douence, Maître de conférence, INRIA, EMN
Gilles Ardourel, Maître de conférence, Université de Nantes

To all my familly members, friends and collègues

Acknowledgements

The work presented in this thesis could not have been possible without the support of many people. Many thanks to my supervisors: Mario Südholt, Rémi Douence and Gilles Ardourel for their timely advice, consultations, encouragement, and critiques throughout the development of this work. Rémi has been an invaluable source of support and guidance all along my work on the thesis.

Many thanks to my small family (my wife and my daughter Rawane), I also would like to thank my Mom and Dad for their encouragement, understanding and caring. They have always been a source of motivation. Without them, I could have never accomplished what I have done today.

Last and not least, I am very grateful to Ascola and Aelos teams for their support of this thesis.

Abdelhakim Hannousse

Abstract

Component based software engineering, or CBSE in short, enables the modularization of concerns in terms of separate software entities called components. Each component provides a set of services and may require services from other components to accomplish its tasks. Components can be assembled in order to construct complex systems. On the other hand, aspect oriented programming, or AOP in short, focuses on the modularization of scattered and tangled concerns that cannot be modularized using regular software entities. Crosscutting concerns are not related to a specific paradigm and CBSE is not an exception. However, current works on CBSE focus only on mapping AspectJ-like concepts into component models missing the particularity of components (*i.e.,* join point model for black boxes) and component systems (*i.e.,* pointcuts defining points in component architectures) and the interferences that may appear when several aspects are woven to a system. In fact, aspect interferences detection and resolution is still a challenge for AOP. In this thesis we contribute by introducing a declarative pointcut language (VIL) for component models, and we provide a formal framework for aspect interference detection and resolution when several aspects are woven to a component system. In our framework we introduce an ADL that extends current ADL(s) with explicit definition of component and aspect behaviors, and aspect weaving rules. Each weaving rule uses our VIL expressions to describe which and where aspects should be woven. We provide a set of transformation rules to obtain the formal specification of components and aspects from the ADL, and we use model checkers for the detection of potential interferences among aspects. For interference resolution, we provide a set of composition operators. Each operator is given with a motivation example, a structure, and a set of applicability rules. The operator structure is described as an abstract form that can be instantiated for any two arbitrary aspects and a set of join points. The list of operators in the catalog is not exhaustive but it can be considered as a first step towards a pattern catalog for aspect interferences resolution. In our framework we adopt the use of Uppaal model checker for its support of template instantiation, local variable declarations, and parameter passing between processes in addition to its support of timing constraints to model real time systems. We illustrate our approach with Fractal component model and a case study: airport wireless access. We define a set of interfering aspects for the example, and we show how our modelization of the system with aspects in Uppaal enables the detection of interferences and how our operators can be instantiated to solve them. Finally, we should mention that our framework is general and can be used for other component models with minimum adaptations.

Résumé

La programmation par composants (CBSE) permet la modularisation des préoccupations en termes d'entités logiciels séparées appelés composants. Chaque composant fournissant explicitement des services en s'appuyant sur des services fournis par d'autres composants. Les composants peuvent être assemblés afin de construire le système global. D'autre part, l'approche aspects (AOP) vise à séparer les préoccupations techniques ou de contrôle (*e.g.,* synchronisation, persistance, contraintes temps réel, etc.) des préoccupations métier ou fonctionnelles. Elle offre un mécanisme de tissage qui permet de fusionner ces deux types de préoccupations afin de construire le système global. Ceci permet une meilleure séparation du code fonctionnel du code non fonctionnel et d'assurer une meilleure maintenabilité du système. Les préoccupations transversales ne sont pas liés à un paradigme spécifique et le paradigme composants n'est pas une exception. Malheureusement, les travaux actuels sur la programmation par composants vise à implenté les concepts d'AspectJ directemment on tel quels dans les modéles à composants ignorant la particularité des composants et les systémes à composants (*i.e.,* points de coupures définissants des points dans les architectures composants) et les interférences entre les aspects qui peuvent apparaissent lorsque plusieurs aspects sont tissés à un systéme. En fait, la détection et la résolution des interférences d'aspects est toujours un défi pour les AOP. Dans cette thése, nous contribuons par l'introduction d'un langage déclarative de points de coupure (VIL) dédié au modèles à composants, et nous fournissent un cadre formel pour la détection et la résolution des interférences d'aspects lorsque plusieurs aspects sont tissés à un système à composants. Dans ce cadre, nous introduisons un ADL qui s'étend ADL(s) actuellent par une définition explicite des comportements des composants et d'aspects, et les règles du tissage et de composition d'aspects. Chaque règle utilise des expressions VIL afin de décrire déclarativement où les aspects vont être tissés. Nous fournissons un ensemble de règles de transformation pour obtenir la spécification formelle des composants et des aspects à partir de l'ADL, et nous utilisons des model checkers pour la détection des interférences possibles entre les aspects. Pour la résolution des interférences, nous fournissons un ensemble d'opérateurs de composition. Chaque opérateur est donnée avec un exemple de motivation, une structure et un ensemble de règles d'applicabilité. La structure d'opérateur est décrit comme une forme abstraite qui peut être instancié pour n'importe quel deux aspects et n'importe quelle ensemble de points de jonture. La liste des opérateurs forme une première étape vers un catalogue pour la résolution d'interférences d'aspects. Dans notre proposition, nous adoptons l'utilisation de model checker Uppaal pour son soutien à l'instanciation des processus, la déclaration des variables locales, et le passage de paramètres entre les processus, en plus de son soutien à des contraintes temporelles pour modéliser les systèmes temps réel. Nous illustrons notre approche avec le modèle de composants Fractal et une étude de cas: l'accès wifi dans un AirPort. Nous définissons un ensemble d'aspects interférant pour l'exemple, et nous montrons comment notre modélisation du système avec les aspects en Uppaal permet la détection d'interférences et de la façon

dont nos opérateurs peuvent être instanciés pour les résoudre. Enfin, il convient de mentionner que notre approche est générale et peut être utilisé pour d'autres modèles à composants avec des adaptations minimales.

Contents

5 Aspects as wrappers on views of component systems architectures 91
5.1 Aspects as wrappers on views . . . 91
5.2 Views definition language . . . 96
5.2.1 The join point Model . . . 96
5.2.2 Syntax of VIL . . . 97
5.2.3 Semantics of VIL . . . 98
5.2.3.1 FPath Query Language . . . 98
5.2.3.2 VIL semantics in FPath . . . 99
5.3 Implementation of VIL in Fractal component model . . . 101
5.3.1 Composable controllers . . . 101
5.3.2 The components of interest belong to the same composite . . 103
5.3.3 The components of interest are scattered in the architecture . 106
5.3.4 Fractal Weaver . . . 109
5.3.4.1 VIL Analyzer . . . 109
5.3.4.2 ADL Transformer . . . 110
5.3.4.3 Julia Config Generator . . . 110

List of Figures

5.1 Performance/Recovery view of the crane 93
5.2 TruckSafety/SaveEnergy view of the crane 94
5.3 CraneSafety view of the crane 95
5.4 RTCrane view of the crane 95
5.5 Wrappers crosscutting phenomenon 96
5.6 Directed labeled graph adopted for component architectures 99

List of Tables

Chapter 1

Introduction

Contents

1.1 The scope of the thesis

Component-based software engineering (CBSE) [Szyperski 2002] focuses on the modularity and the reusability of software systems. Following CBSE, the behavior of a system is divided into different parts and implemented by separate and reusable entities named components. Components can be assembled in order to obtain the complete and desired application. This considerably reduces software complexity and enhances the understandability, the reusability and the maintenance of complex software systems. On the other hand, aspect-oriented programming (AOP) [Kiczales 2001a] focuses on the separation of concerns in general and the modularity and the reusability of non-modular (*a.k.a.,* non-functional, extra-functional) properties of software systems in particular (*i.e.,* persistence, synchronization, real-time constraints). Those properties are scattered and tangled over several entities and cannot be encapsulated in only one entity. AOP provides a set of mechanisms to deal with such properties: *join point models*, *pointcut languages*, and *weaving strategies* (*i.e.,* compile time, load time, or runtime weaving). AOP is considered as a paradigm independent approach that can be implemented for functional [Dantas 2008], procedural [Coady 2001], object-oriented [Kiczales 2001b] or even component-based [Suvée 2003] paradigms. Putting CBSE and AOP together is a promising approach that ensures better modularity and reusability of software components: when aspects are kept modular after weaving, component and aspect modularity and reusability are preserved. The main challenge for CBSE to support AOP, is twofold. First, how to model aspects in a component model, in other words, should aspects be implemented as regular architectural elements (symmetry approach) or new elements should be defined in CBSE (asymmetry approach)? [Suvée 2006]. Second, aspect interferences is still a challenge in AOP, so

what happens when several aspects are woven to the same component system?, how there interferences can be detected and solved?. Current works on the integration of AOP into CBSE are limited to: (1) map object-oriented pointcut languages into component models, (2) the proposed implementations are adhoc solutions and cannot be generalized for other component models, and (3) the proposed approaches do not provide an effective support for aspect interference detection and resolution.

1.2 Contributions

In this thesis we focus on the *aspectualization* of component models in general. This includes the definition of a pointcut language dedicated for component models, modelling aspects in an abstract way without much care about their implementation (*i.e.,* symmetry or asymmetry approach). In addition, we provide a support for interference detection and resolution. During our exploration of existing component models, we observed that hierarchical component systems can be reconfigured in different ways, each of which defines a point of view of the architect on the system. On the other hand, aspects are scattered over different components, and in order to preserve the modularity of aspects, all the components of interest to an aspect must be encapsulated together. In our proposal we get benefit of configuration variation of component systems and we define a view (a system reconfiguration) for each aspect. Fulfilling this requirement, we define a generic and a declarative pointcut language named VIL (for VIews definition Language). We give its syntax and semantics in terms of generic concepts of component models to preserve its component model independence. In addition, we provide a unified formal model for software components and aspects, and we use Uppaal model checker for the detection of potential interferences among aspects. For interference resolution, we provide a set of composition operators as a catalog of patterns. Inspired from the GoF patterns [Gamma 1995], each operator is illustrated with a motivation example, applicability circumstances, structure (as a state machine), and a semantic (based on the actions taken by aspects and the intercepted join points). We evaluate our proposal by implementing the VIL language, the composition operators and the formal modelling process for Fractal component model [Bruneton 2004]. In addition, we illustrate the proposal with a case study: a wireless access in an airport [Šery 2007, Adamek 2007]. The following section, introduces an illustrative example that we use all along the thesis chapters in order to help the reader understanding the introduced concepts of our proposal.

1.3 Illustration Example: Crane System

Our example is a revised version of the one given in [Bergmans 1996]. It describes a software controller of a crane that can lift and carry containers from arriving trucks to a buffer area and vice versa. The crane system is composed of an engine that moves the crane left to the truck and right to the buffer area, a mechanical arm

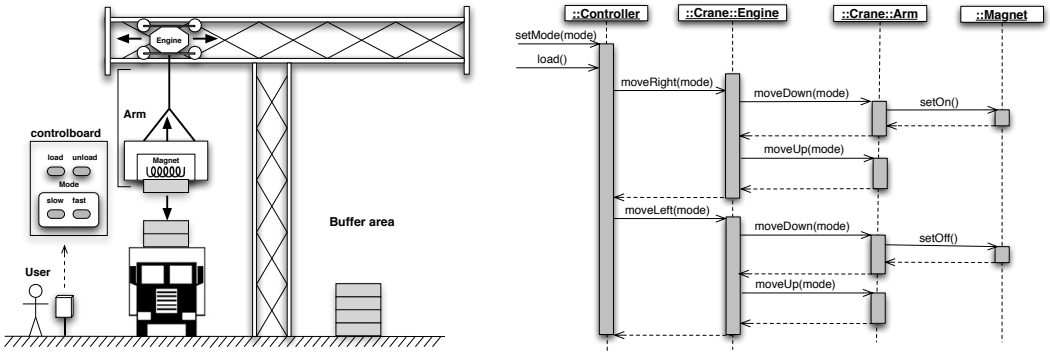

Figure 1.1: Running example: (a) overview, (b) loading process scenario

that moves up and down, and a magnet for latching and releasing containers by activating and deactivating its magnetic field. The engine and the arm may run in two different modes: *slow* and *fast*. Users interact with the crane using a control board. The control board allows users to choose a running mode (*i.e.,* fast or slow) for the crane and to start loading or unloading containers. Figure 1.1(a) and figure 1.2 depict, respectively, a schematic overview and a possible component architecture of the system. Figure 1.2 models the crane system as a component architecture with

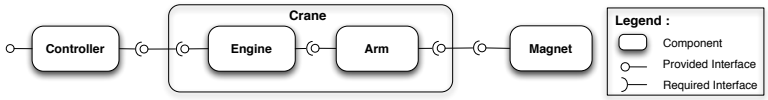

Figure 1.2: Crane system architecture

three main components: *controller, crane* and *magnet*. The controller component provides an interface that permits to set the running mode of the crane and start loading and unloading containers. Upon receipt of user commands, the controller component transforms those commands into signals and requires the crane to act following those signals through its required interface. The crane component is a composite of an *engine* and an *arm* components. The engine component provides an interface that permits to move the crane left and right following a running mode and requires an interface to call the arm to move up and down. The arm, in turn, provides an interface for moving up and down following a running mode and requires an interface to ask the magnet to latch or release a container. Finally, the magnet component provides merely an interface for latching and releasing containers. Figure 1.1(b) shows the UML sequence diagram of loading a single container. The process of loading a container starts when a user sets the running mode for the crane and presses the load button on the control board. These two actions are transformed into calling *setMode* and *load* services, respectively, on the provided interface of the controller component. When the controller component receives a *load* service call, it requires the engine component to move right by calling *moveRight* on its required interface. Upon receipt of *moveRight* call, the engine does the action and

requires the arm to move down by calling *moveDown* service. The arm accepts the call, moves down and asks the magnet to latch a container from the buffer area by calling *setOn* service on the provided interface of the magnet. When the container is latched, the engine calls the arm to move up throwing a *moveUp* call. When all this done, the controller component requires the engine to move left to the truck by calling *moveLeft* service. The engine receives the call, asks the arm to move down which in turn asks the magnet to release the latched container by calling *setOff* service. This basic crane system is later on enhanced by forcing it to fulfil a set of new requirements. Each one of those requirements is modeled as an aspect to be woven to the system:

Performance:
Enforce the crane to move fast, whatever was the running mode chosen by the user, when the arm is not carrying a container. This considerably improves the general response time of the crane.

Recovery:
Return both the engine and the arm to their stable position in the middle whenever an undesirable sequence of actions is captured. This ensures the viability of the crane system.

Truck Safety:
enforce the arm to move slow, whatever was the running mode chosen by the user, when the crane is loading a container on the truck. This ensures the safety of both the truck and the containers.

Save Energy:
enforce the arm to move slow, whatever was the running mode chosen by the user, after carrying a given number of containers. This ensures a better energy consumption of the crane.

Crane Safety:
Ignore user commands when the temperature of the engine or the arm reaches a critical value. This ensures the safety of the crane devices.

Real-Time:
Check whether loading/unloading containers is achieved in t_{speed} ($\leq t_{speed}$) *time. If it is not the case, the arm must be moved up and all the subsequent requests must be refused.*

1.4 Thesis structure

This thesis is divided into two parts: background and contributions. In the background, we first introduce and evaluate current works on AOP (**Chapter 2**). Second, we overview current works on aspectualized component models highlighting their strengths and weaknesses and the points that should be considered for a generic

approach to aspectualize component models (**Chapter 3**). The second part is divided into three chapters: we introduce our proposed generic and declarative pointcut language for component models (VIL) and we show how it can be implemented in Fractal component model (**Chapter 4**). Then, we describe our formal model for the detection and the resolution of aspect interferences in component models (**Chapter 5**). Finally, we illustrate our approach with a case study (**Chapter 6**) and we conclude the thesis (**Chapter 7**). Figure 1.3 gives a reader a general idea about our views-based approach for aspectualizing component models and gives a reader a suggestion of how to read the thesis according to his/her purpose. The structure is modeled as a component architecture with different views. Each view is presented as gray shape and annotated with a title in a white box. A binding of two components c_1 and c_2 indicates that c_1 requires concepts and details given in c_2. Finally, we use a dashed box to indicate a shared component.

1.5 Published papers

1. **A.H. Hannousse**, Rémi Douence, Gilles Ardourel, *Static Analysis of Aspect Interaction and Composition in Component Models*, In Proceeding of the 10th International Conference on Generative Programming and Component Engineering, GPCE'11, Portland, Oregon, USA, ACM, 2011

2. **A.H. Hannousse**, Rémi Douence, Gilles Ardourel, *Composable Controllers in Fractal: Implementation and Interference Analysis*, In Proceeding of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA'11, Oulu, Finland, IEEE CS, 2011

3. **A.H. Hannousse**, Gilles Ardourel, Rémi Douence, *Views for aspectualizing component models*, In Proceedings of the Ninth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS'10, pp. 21-25, Rennes, France, 2010.

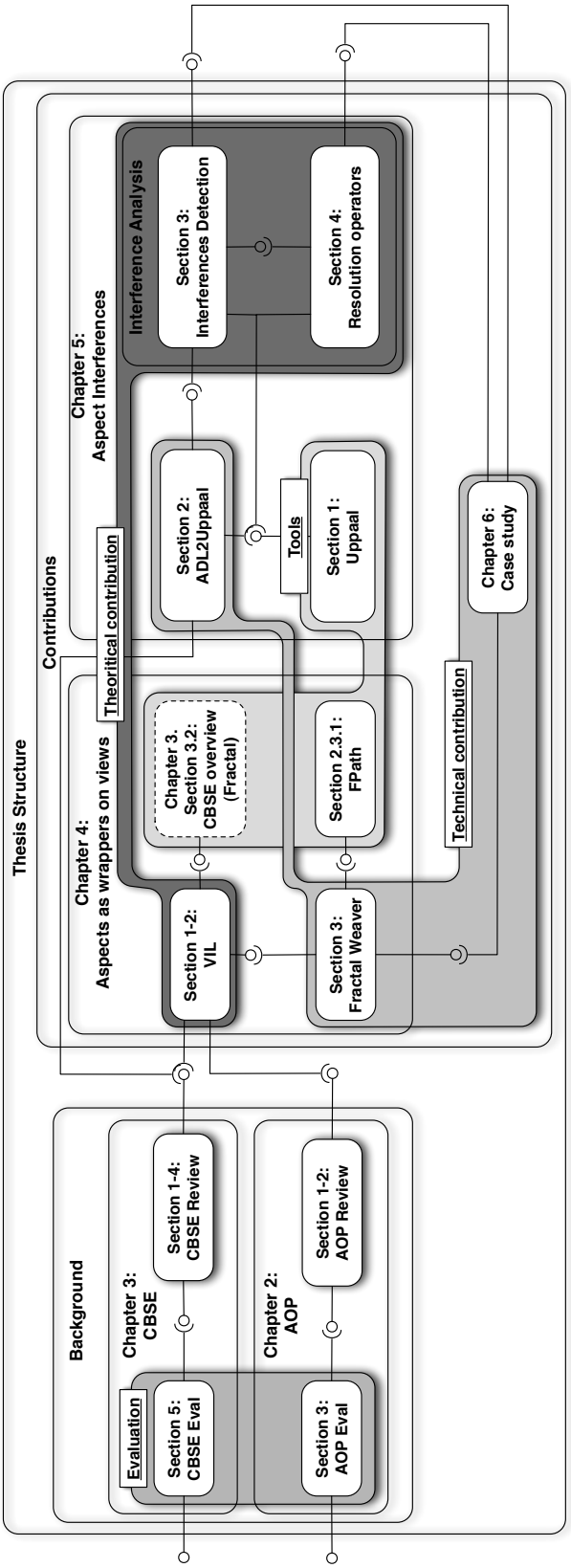

Figure 1.3: Views on the thesis structure: how to read the thesis

Part I

Background

Chapter 2

Aspect Oriented Programming and Aspect Interference Issue

Contents

2.1 Overview of AOP

Object-Oriented Programming (OOP) [Meyer 1997] abstracts and encapsulates concerns into separate entities called classes. It provides a set of mechanisms to rely and compose them: inheritance, aggregation, and polymorphism. However, some concerns fail to be encapsulated into single and separate classes and, instead, they spread over multiple classes. These particular concerns are known as *crosscutting concerns*. Aspect Oriented Programming (AOP) [Kiczales 2001a] provides a mechanism to cope with the encapsulation of concerns in general and crosscutting concerns in particular. It provides a mechanism to encapsulate concerns, crosscutting or not, into separate and independent modules. It comprises techniques as well as tools for composing these modules to get the final program. Here we give the basic tenets of AOP; first we introduce the basic concepts common to all the AOP approaches:

Join point: is a particular point in the control flow of a program. Join points are caught when programs are under execution. Crosscutting concerns interact

only with other concerns at these points. In object oriented paradigm, these points can be a method call, an attribute access, or an object instantiation.

Pointcut: is a predicate that matches a set of join points. A pointcut joins a set of join point expressions with logical operators. An AOP model provides an expressive pointcut language that specifies the set of required join points in a declarative style.

Advice: implements a whole or a part of the behavior of a crosscutting concern. An advice is executed when pointcuts matching succeeds. An advice is executed before, after or instead of the matched join point(s).

Aspect: is a modularization unit that encapsulates a crosscutting concern. An aspect specification includes pointcuts and advices definition.

Weaving: is a mechanism that merges aspects modeling crosscutting concerns with the rest of modules modeling regular concerns. This is necessary to get the complete program with all the concerns needed. The result program is called *aspectualized program* due to the integration of aspects into its basic program. A weaving is an automatic process performed by tools called *weavers*. Aspects can be weaved at compile time, post-compile time, load time or runtime. Each mode of weaving has its pros and cons depend on a set of properties such as: the availability of the source code of the basic program, the performance of the process, etc. Note here, that it is possible that a weaver implements more than one weaving mode. The reverse process of weaving is called *unweaving*. It is useful to ensure dynamic reconfiguration of software systems.

Figure 2.1 illustrates the process of weaving and unweaving aspects. In the figure, a concern is modeled as a set of squares, each of which represents a unit of a module (*e.g.,* method, statement, etc.). While colored squares refer to aspect units, white squares refer to base program units. A pointcut is modeled as a set of integer values indicating the positions where aspect units should be injected to the base program. For example (0,4) indicates that the aspect unit should be injected at the first and the fourth positions of the base module. The result system after weaving is a mixture of a white and colored units. The unweaving process is possible if we keep track of the positions of the already woven aspects.

The same aspect may affect several program modules and several aspects can be woven to the same base program. In the former case one or more instances of aspects may be created. This is useful when shared variables are considered. In the latter case, each aspect extends the base program with new features; this generally leads to what is called *interaction* between aspects. In some cases interactions are natural and desirable, but in many cases, interactions are undesirable because they lead to an unexpected behavior of the woven program. We call this latter kind of interactions *interferences*. Actually, one of the most challenging issues of AOP implementation models is how to weave several aspects without potential interferences. The above mentioned concepts of AOP are modeled differently according

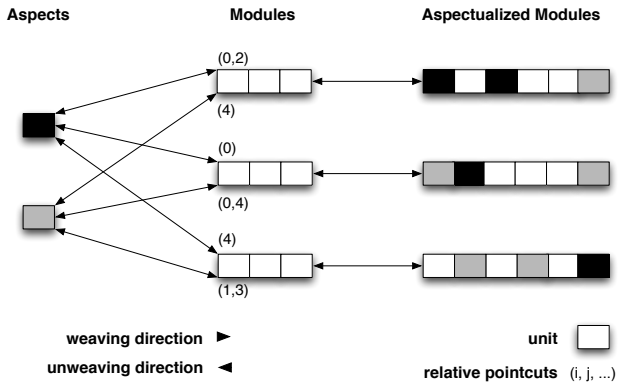

Figure 2.1: AOP weaving and unweaving process

to each implementation model, and aspect interferences are managed at different granularity levels. In the following, we discuss three implementation models and we show how the above concepts are modeled and how interferences are managed.

2.1.1 AspectJ

AspectJ [Kiczales 2001b] is the reference implementation of AOP in Java. Aspects are modular units encapsulating crosscutting concerns with explicit definition of pointcuts and advices. AspectJ enables static and dynamic crosscutting. By static crosscutting, an aspect defines additional attributes and methods to existing Java classes, and also enables the declaration that an existing class implements new interfaces or extends new other class. By dynamic crosscutting, AspectJ enables the execution of additional behaviors (advices code) at well-defined points in the execution of a program (pointcuts). AspectJ provides a declarative pointcut language for object-oriented programming; a pointcut captures a set of join points using a set of predefined terms and composes them using logical operators. The pointcut language includes: calling and executing methods or constructors, setting and getting attributes and more advanced pointcuts such as `cflow(jp)` that captures each join point in the control flow of *jp* including *jp* itself. In [Allan 2005, Douence 2006], AspectJ pointcut language is extended with *tracematch* pointcut where advices' execution is based upon the execution history of a program. With tracematches, regular patterns of events are specified, and aspect advices are executed when the execution trace of a program matches one of these patterns at runtime. The advices in AspectJ are of three main types: *before*, *after* and *around* that specify that the aspect behavior, implemented by the advice, is executed before, after or instead of the captured join points, respectively. Listing 2.1 describes the `saveEnergy` aspect in AspectJ. In the listing, two pointcuts are defined `left` and `right` (line 5-6); the former captures calls to all the methods whose names end with *load* in the `Controller` class and the latter captures all the methods whose names start with *move* in the `Arm` class. Two advices of type *before* are defined for each pointcut. The former (line 8-10) increments the number of carried containers, the latter (line

12-14) checks whether the number of carried containers reaches 100. If it is the case, it calls a local method `setSpeedParameter` to change the speed parameter of the captured join point to `SLOW` value.

```
1 aspect saveEnergy {
2   declare precedence : truckSafety, saveEnergy.
3   private int NbContainer =0;
4
5   pointcut left() : call(* Controller.*load(*));
6   pointcut right() : call(* Arm.move*(*));
7
8   before() : left() {
9       NbContainer++;
10  }
11
12  before() : right() {
13    if (NbContainer>100) setSpeedParameter(SLOW);
14  }
15
16  void setSpeedParameter(Speed s) {
17    // code not shown here
18  }
19 }
```

Listing 2.1: The `saveEnergy` aspect in AspectJ

For interference management, AspectJ defines a set of precedence rules to resolve the execution order of multiple advices at the same join points. A precedence order is explicitly defined for aspects by either sub-classing (*i.e.,* an aspect is declared as a subaspect of another) or using `declare precedence` keyword. Another implicit precedence rules are inferred from the type of advices and their declaration order. For example, the advice that appears first lexically inside an aspect is the one who executes first. In the crane example, the `truckSafety` aspect advices should be executed before the advices of the `saveEnergy` aspect, this can be specified using `declare precedence` statement as shown in Listing 2.1 line 2.

In AspectJ, aspects can be either singleton or can be instantiated per object and even per control flow. In the former case, an aspect is instantiated for every object triggering a join point (*i.e.,* `perthis`) or for every target object of a join point (*i.e.,* `pertarget`). In the latter case, an aspect instance is created for every control flow beginning at a join point picked out by a pointcut parameter.

AspectJ weaver tool *ajc* supports bytecode weaving where the code of advices is added to class files at compile time, post-compile time or at load time; the weaver should only be parametrized with the required weaving mode.

2.1.2 Composition Filters

Composition Filters (CF) [Aksit 1992] provides a support of AOP to the object oriented paradigm. In CF an object is encapsulated with a wrapping layer called *interface* that intercepts and handles incoming and outgoing calls by means of filters. Incoming calls pass through a sequence of filters called *inputfilters* and outgoing

calls pass through a sequence of filters called *outputfilters*. A filter corresponds to an aspect that wraps an object and decides to accept or reject calls. Each filter has a type which in turn has a well-defined semantics of calls acceptance and rejection. CF supports a set of predefined filter types and allows the definition of new filters when they are needed. The current provided filter types are: *Wait* (for calls synchronization), *RealTime* (for real time properties management), *Error* (for error handling), *Meta* (for message reification), *Substitute* (for substituting messages properties), and *Send*, *Dispatch* (for message delegation in input and outputfilters, respectively). These filters are orthogonal which means that they are independent and they can be easily composed by forming a chain. Filters are reusable entities that can be instantiated by specifying their elements. A filter element corresponds to a pointcut in AOP, if the intercepted message matches a filter element, the message is accepted, otherwise it is matched to the next filter element; if it does not match the last filter element, the message is rejected. The acceptance and the rejection actions depend on the filter type. For example, a filter of *Meta* type accepts a message by sending it as a parameter of another message to a named object, and rejects a message by allowing it to pass to the next filter in the chain. Besides input and output filters, the interface layer encapsulates a set of objects called *internals*; these objects implement new functionalities that have to be added to the encapsulated object named *inner object*. The interface layer also refers to a set of external objects that are objects communicating with the inner object. Figure 2.2 shows the architecture of the CF object model.

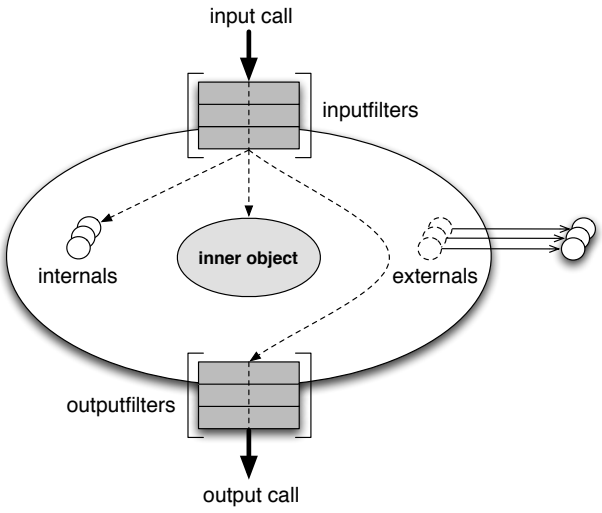

Figure 2.2: Composition Filter wrapping mechanism

Originally, the CF is designed to model aspects that crosscut a single object. In later work [Bergmans 2001], the approach is generalized to model aspects that crosscut multiple objects using *superimposition*. The superimposition defines a set of filters and maps them to multiple objects. However, filters are only composed sequentially following their declaration order. Listing 2.2 illustrates the implemen-

tation of the `saveEnergy` aspect in CF. The aspect defines two interface layers `leftModule` and `rightModule`. The former uses a `Meta` and a `Dispatch` as inputfilters and the latter uses a `Substitute` and a `Send` as outputfilters. The `Meta` filter `met` (line 9) reifies each incoming call whose name ends with `load` and passes it to an external object (`saveEnergy`) to execute its `maintainNbContainer` method. This latter increments the number of carried containers (line 39) and proceeds the call by calling the predefined `fire` method (line 40). The `Dispatch` filter `dis` (line 10) accepts all the calls to the inner object instance. The `Substitute` filter `sub` (line 19-22) changes the parameter value of the `moveLeft` and `moveRight` messages to `SLOW` whenever the number of carried containers reaches a threshold number (*e.g.,* 100 in this case); this is checked by calling the `isThresholdNbReached` method of the external object (line 42-44). Note that the `Dispatch` and `Send` filters should be used as the last filters in inputfilters and outputfilters sets, respectively, to transmit messages to their target objects. The superimposition part (line 26-33) specifies a set of selectors that specify a set of objects that should be wrapped with a specific interface layer. In the listing, two selectors are defined `left` and `right` (line 28-29). The former includes all the instances of the `Controller` class while the latter includes all the instances of the `Engine` class. The filtermodules part (line 41-42) states that the `leftModule` and `rightModule` wrap all the objects defined by the `left` and `right` interface layers, respectively.

```
1  // interface layers part
2  concern SaveEnergyAspect {
3    filterinterface leftModule {
4      externals
5        saveEnergy: SaveEnergy;
6      methods
7        maintainNbContainer(Message);
8      inputfilters
9        met: Meta = {True => [*.*load(..)] saveEnergy.maintainNbContainer}
10       dis: Dispatch = {inner.*}
11   };
12
13   filterinterface rightModule {
14     externals
15       saveEnergy: SaveEnergy;
16     conditions
17       saveEnergy.isThresholdNbReached;
18     outputfilters
19       sub: Substitute = {
20         isThresholdNbReached => [*.moveLeft(..)] *.moveLeft(SLOW),
21                                 [*.moveRight(..)] *.moveRight(SLOW)
22       }
23       send: Send = {*.*};
24   };
25
26   superimposition {
27     selectors
28       left = {*=Controller};
29       right= {*=Engine};
30     filtermodules
31       left <- leftModule;
32       right <- rightModule;
33   };
```

```
34
35 // implementation part
36   class SaveEnergy {
37     private int NbContainer = 0;
38     void maintainNbContainer(Message m) {
39       NbContainer++;
40       m.fire();
41     }
42     boolean isThresholdNbReached() {
43       return NbContainer>100;
44     }
45   }
46 }
```

Listing 2.2: The `saveEnergy` aspect in CF approach

2.1.3 Hyper/J

Hyper/J [Tarr 2000] implements AOP following the multi-dimensional separation of concerns approach (MDSOC) [Tarr 1999, Ossher 2001]. In MDSOC, a software system is divided into different units called *concerns*. A concern describes a part of interest or a role of a software unit (*e.g.,* class, method, attribute). Different types of concerns may exist for the same software system. For example, the base program of an application is a concern, and all the aspects to be applied to that application can either be of the same concern type or different concern types according to their roles. In our crane example, all the primitive components constitute one concern type called *base* (one dimension), while the proposed aspects can be considered as one concern type (another dimension) named *optimization*. Affecting software units to concerns is explicitly specified in a separate module called *hyperslices*. Concerns of different or the same type interact with each other and can be easily managed and composed using a set of weaving operators named *relationships*. Hyper/J provides a collection of relationships such as: `mergByName` and `bracket`. The former, indicates that units of the same name in different concern types (dimensions) are merged and composed together and form a new unit. The latter indicates that a set of units should be executed before or after other units. The weaving strategy supported by Hyper/J for merging units is a post-compile time, where the bytecode of the composed units are merged.

```
1 hyperspace craneSystem
2   class User; class Controller; ...; class SaveEnergy; class TruckSafety;
3   class Performance;
4
5 hyperslices
6   class User : Feature.base
7   class Controller : Feature.base;
8   ...
9   class SaveEnergy : Feature.optimization;
10  class TruckSafety : Feature.optimization;
11  class Performance : Feature.optimization;
12
13 hypermodule ExtendedCraneSystem
14   hyperslices
15     Feature.base,
```

```
16      Feature.optimization,
17    relationships
18      bracket Feature.base."*load"
19      before Feature.optimization.TruckSafety.setLoadingState,
20      before Feature.optimization.SaveEnergy.maintainNbContainers;
21
22      bracket Feature.base.moveUp
23      before Feature.optimization.SaveEnergy.checkForSpeed;
24
25      bracket Feature.base.moveDown
26      before Feature.optimization.TruckSafety.checkForSpeed,
27      before Feature.optimization.SaveEnergy.checkForSpeed;
28
29      bracket Feature.base.moveLeft OR Feature.base.moveRight
30      before Feature.optimization.Performance.checkForSpeed;
31
32      bracket Feature.base.set*
33      before Feature.optimization.Performance.setMagnetState;
34  end hypermodule;
```

Listing 2.3: The crane example in Hyper/J

Listing 2.3 illustrates the crane example in Hyper/J. In the listing, the set of software units are declared in a hyperspace module (line 1-3); in the example this includes all the classes implementing both components and aspects. Software units are mapped to concerns in a hyperslice module (line 5-11). As mentioned above, components are mapped to the *base* concern type while our aspects are mapped to the *optimization* concern type. The composition of concerns is defined in a hypermodule (line 13-34). In the hypermodule, the set of all the concern types to be composed is determined with the hyperslices keyword (18-20), and the composition rules are defined within the relationships keywords (line 21-34). In our example, only the `bracket` composition rule is used to order the execution of methods of different concern types. For example, the bracket rule at line (22-24) indicates that whenever a method whose name ends with `load` (this includes `load` and `unload` methods of the `Controller`) of a base concern type is invoked, two methods of the optimization concern type should be executed before; that are: `setLoadingState` of the `truckSafety` concern and `maintainNbContainers` of the `saveEnergy` in such order. The other bracket rules are interpreted similarly.

2.1.4 Evaluation

The above models implement the AOP tenets in general. They show how aspects, pointcuts and advices are modeled and specified within language constructs, when aspects are woven to base systems (*i.e.,* compile-time, runtime, etc.), and how several aspects are ordered. Each of the above models has its strengths with respect to AOP: AspectJ provides an expressive and a declarative pointcut language for object-oriented models, especially with trace matches and control flow pointcuts support, CF provides a set of reusable filter types encapsulating aspect semantics such as synchronization and real-time constraints, and Hyper/J provides a symmetry approach with a minimum language constructs to define aspects and offers a set of predefined composition rules to compose their advices. However, these three

models share a set of limitations: they only support sequential ordering of aspects and do not provide any support of aspect interferences analysis. In the following we overview a set of works dedicated to aspect interferences detection and resolution.

2.2 Aspect Interferences

Aspects interact in different ways even if they are orthogonal. When several aspects needed to be applied to the same system, undesired interactions may appear. We call such undesired interactions *interferences*. Many works are dedicated to detect and/or solve interferences among aspects. The proposed approaches can be divided into two categories: *syntactic* and *semantic* based approaches.

2.2.1 Syntactic-Based Approaches

The focus of syntactic-based approaches is limited to analyze aspects sharing join points or updating common variables of base systems. In this section, we overview some relevant works to this category: Durr et al [Durr 2007, Durr 2008] propose a resource-base model for interferences detection at shared join points. The proposed approach models each join point as a resource manipulated by different aspect advices. A set of operations, on those join points, are defined such as: *read* and *write*. The conflicts treated are of two kinds: *data* and *control flow* conflicts. Data conflicts are related to modifying a resource properties, while control flow conflicts are related to advices actions on resources (*i.e.,* proceed or skip). For example, a sequence (*write;write*) of the same join point parameter is a source of data conflict. While, a sequence (*skip;proceed*) of the same join point is a source of control flow conflict. For conflicts detection, advice actions performed on each join point are tracked and a set of conflicting rules are defined. A conflict rule is a sequence of actions. When a conflicting rule matches a part of the actions trace on a join point, a conflict is reported to the user. For showing statically the different sequences of advices actions performed on shared join points, a *message flow graph* is generated. Each node of the graph represents a program element to be evaluated (*i.e.,* join point, pointcut, data variables, etc.), while each path in the graph represents a possible sequence of actions performed by advices at a shared join point. Arcs in the graph are labeled with resource operations (*e.g.,* read or write) and advice method names or actions (*i.e.,* proceed or skip). Figure 2.3 shows the message flow graph showing the interaction between the `saveEnergy` and the `truckSafety` aspects. Only the shared join points are presented in the graph. Normal font labels are advice method names, pattern matching or condition evaluation. Italic and underlined labels are resource operations, while bold font labels are advice actions. From the graph, a conflict is detected for the `moveDown` call join point, because of the sequence: `args[0].write;args[0].write` in the path: both aspects change the same parameter.

Since traces cannot be detected statically, a runtime process is provided by the approach. The runtime process uses an abstract virtual machine AVM. The virtual

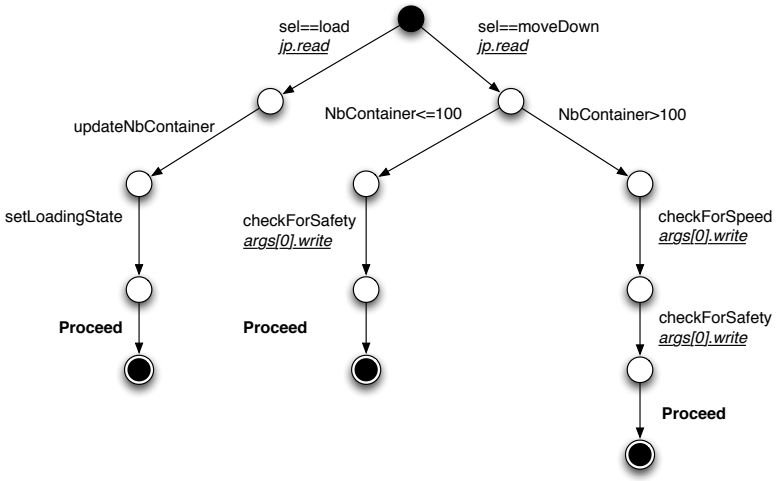

Figure 2.3: An excerpt of the message flow graph representing the crane example showing the interaction between `saveEnergy` and `truckSafety` aspects

machine keeps track of the different advice actions and resource operations applied to each join point at runtime. When the saved trace matches one of the conflicting rules, a warning is reported to the user. The given approach is applied to the composition filters (CF) and implemented to its Compose* toolset [de Roo 2008]. The approach has two main drawbacks: first, all the conflicting rules should be explicitly determined by the user which is a tedious and error prone task. Second, take for example, the conflicting rule: `args[i].write; args[i].write`, it does not always refer to a conflict, the interaction between the `saveEnergy` and the `truckSafety` aspects is a counter example: when both are applied sequentially, they both overwrite the parameter value of the `moveDown` call but there is no interference between them, and hence, false positives can also be reported within the approach due to lack of knowledge about the intent of each aspect. Finally, the approach supports user defined advice actions, however, resource operations actions should be abstract as possible, otherwise, a global awareness and knowledge of all the aspects to be woven is needed. This threatens the separation of concerns principle and hardens the update of these properties whenever other aspects needed to be applied.

Marot et al. [Marot 2009] tackle the above limitations by replacing global conflicting rules by local assumptions called compositional intentions. Compositional intentions are defined for each advice and describe assumptions about the actions of other advices on shared join points. Each compositional intention has a *type* and a *behavior description*. The compositional intention type describes when the advice should be executed with respect to other advices. Besides *before* and *after*, *ignored* is used to indicate that the advice should be ignored when the behavior description part is matched by other advices. A behavior description is a conjunction and/or a disjunction of *action predicates*. An action predicate denotes the action

of an advice. An advice may proceed a join point (*proceed*), skip it (*NoProceed*), make multi proceed calls (*MultiProceed*), or substitute the join point properties (*proceedWithSubst*). These are more precise control flow actions than those defined in [Durr 2007, Durr 2008]. Data flow predicate actions are also supported by the approach but in a more general form: only *read* and *write* of variable instances are supported. A compositional intention of an advice states that, if the behavior description is matched by another advice, according to the type, the advice in question is executed before, after that advice or its execution is completely ignored. For example, the `saveEnergy` aspect's `maintainNbContainer` advice can be associated with the following compositional intention.

```
1 Ignored: NoProceed();
2 After: MultiProceed();
```

Listing 2.4: Compositional intention of the `maintainNbContainer` advice

This compositional intention states that, if the join point (*i.e.,* `load` or `unload` method calls in this case), is skipped by another advice, the `maintainNbContainer` advice is ignored and hence the number of carried containers is not incremented (line 1); and whenever the join point is called, the advice is executed (line 2). This ensures the correct composition of the `saveEnergy` aspect with an aspect that skips the call of `load` or `unload` methods to the `Controller` such as the `craneSafety` aspect. Remember that the `craneSafety` aspect skips all the user commands when the temperature of the crane exceeds a threshold value. In the approach, conflicts are detected at runtime: an assumption of an advice is compared with all the other advices, if the assumption of the advice matches the assumption of another with incompatible type, an error is reported to the user. However, the proposed compositional language is not expressive enough to describe all the kinds of interferences accurately. Take for example, the `truckSafety` and the `saveEnergy` aspects; their corresponding `checkForSpeed` advices (`checkForSpeed` advice changes the speed parameter value of the call to `SLOW` when a correspondent predicate holds) are defined with the following compositional intention:

```
1 Before: Read(thisJoinPoint, args(0), speed) ∧
2         Write(thisJoinPoint, args(0), speed);
```

Listing 2.5: Compositional intention of the `checkForSpeed` advice

This compositional intention states that the `checkForSpeed` advice is assumed to be executed before any reading or writing of the first join point argument `speed` (the first argument of the call `args(0)`). However, when both `saveEnergy` and `truckSafety` aspects are composed, a conflict is detected, because their corresponding `checkForSpeed` advices assume to be executed before each other. This is correct when they give a different value to the speed argument. In the example, they give it the same value `SLOW`. Thus, an error is reported where there is no a real conflict and

the change of the parameter value cannot be expressed with the current language: `write` action does not mean that the value is changed.

Douence et al. [Douence 2002] address aspect interaction problem at shared join points. In this work, a formal expressive crosscut language (EAOP) is defined and a linguistic support for conflict resolution is proposed. The proposed framework is based on observable events. The following listing describes the `truckSafety` aspect of the crane system following EAOP syntax: the aspect has two parts; the first (line 1) defines a *crosscut* that matches the `load` method call event, and an *insert* that executes the `setLoadingState` method and proceeds the call. The second part defines a *crosscut* that matches the `moveDown` method call event, and an *insert* that executes the `checkForSpeed` method and proceeds the call. The μa is used for recursion to say that an aspect starts in a state a, waits for one of the events (`load` or `moveDown`), executes the corresponding advice code, proceeds the call (`proceed` keyword) and returns to the state a.

```
1 truckSafety = μa (load() ▷ setLoadingState() proceed; a) ||
2               μa (moveDown(..) ▷ checkForSpeed() proceed; a)
```

Listing 2.6: The `truckSafety` aspect in EAOP

On the interaction point of view, this work focuses on aspects sharing join points. Two forms of independence are distinguished: *strong independence* and *independence w.r.t. a program*. The former occurs when two aspects can be expressed as a single aspect without any overlap of their crosscuts, whereas the latter is relative to a given base program by checking all its possible observable execution traces. For each case, an algorithm is provided to check whether two aspects are independent. An interaction between aspects is detected when their crosscuts match the same join point. The weaving process is modeled by analyzing aspect rules and determining the set of rules applicable to the current join point and to the next join point in the trace.

The proposed solution is a set of parallel operators indicating to the weaver how to compose aspects on conflicting points: execute them in sequence, execute only one and ignore the other, etc. For example, the `saveEnergy` and the `truckSafety` aspects are composed sequentially as follows: `saveEnergy` $\|_{seq}$ `truckSafety`. One interesting feature of the approach is the definition of scopes for aspects. A scope definition controls the visibility of aspects; this ensures that non-terminating weaving never occurs: an aspect does not match join points coming from arbitrary other aspects. The work is later extended to stateful aspects [Douence 2004] and for concurrent aspects [Douence 2006]. In the former work, the model is extended with inter-crosscut variables. While the latter provides a support of concurrent aspects, models the woven program as FSP processes and checks properties with LTSA.

2.2.2 Semantic-Based Approaches

Semantic-based approaches do not focus on shared join points and common manipulated variables. They consider the behavior of the aspects to be woven. The approaches of this category can be divided into two subcategories: *modular* and *non modular* approaches. Modular approaches detect behavioral conflicts of aspects independently of any base program. While non-modular approaches detect semantic conflicts with respect to a base program. In the following we overview some works of both categories.

2.2.2.1 Modular Approaches

Katz et al. [Katz 2008] propose the use of LTL (linear temporal logic) formulas to define a set of *assume-guarantee* properties of aspects. The assume properties are general properties any base program should satisfy otherwise the aspect cannot be woven to that program. The guarantee properties are satisfied by the program after weaving the aspect. A pairwise check is performed by the approach: two aspects are interference-free if when they are woven to a base program satisfying their *assumption* properties, their *guarantee* properties should be satisfied after weaving. For formal verification, Maven model checker is used for automatic verification of properties. With Maven, an aspect is checked with respect to its specification. In an extended work with Goldman [Goldman 2010], state machines are used to model the base program, the aspects, and the woven system. The weaving process is implemented by inlining the aspect state machine directly in the base system state machine. However, the approach only focuses on weakly invasive aspects and the precedence interaction between aspects. Moreover, when precedence does not solve the interference, the approach does not provide any solution. In fact, this work is proposed to cope with the limitations of Krishnamurthi et al. [Krishnamurthi 2007] proposition, where a state machine is defined for each advice, and focus on treating aspects not modifying data variables of base systems.

Kniesel [Kniesel 2009] defines a predicate-based formal model that describes aspects in particular and programs in general as a set of *conditional transformations* (CTs). A conditional transformation (CT) is defined as a pair of formulas: *precondition* and *transformation*. Preconditions are predicates that a base program should satisfy, otherwise the transformations (*i.e.,* advices) become inapplicable to the program. Transformations are of three kinds: adding, deleting elements (*i.e.,* attributes, methods) to/from the base program or creating new element identities. Aspect weaving is modeled as applying the CTs modeling an aspect to the base program. The effects and the postconditions of each CT are automatically derived from its precondition and transformation. While effects inform the direct changes on the base program after executing a CT transformation, postconditions tell what can be deduced from the precondition of a CT. Effects and postconditions are used to detect potential interactions between aspects (CTs) in a modular way (without refer to a concrete base program): an interaction between two aspects A_1 and A_2 is detected when the execution of A_1 affects the execution of A_2

positively or negatively. Positive effect is to satisfy the precondition of A_2, which was previously not satisfied, after the execution of the transformation of A_1. Negative effect is to falsify the precondition of A_2, which was previously true, after the application of the A_1 transformation. Affecting one aspect positively is called *triggering* while affecting one aspect negatively is called *inhibition*. For interference detection, a *weaving interaction graph* is designed. The graph nodes denote the different transformation actions (advices), and edges between two nodes indicate an interaction detected between the transformations of the nodes (triggering or inhibition). Figure 2.4 shows the graph representing the interaction between the advices of the `saveEnergy` and `craneSafety` aspects of the crane example. The graph shows that the `maintainNbContainer` advice triggers the `checkForSpeed` advice. That is because the action (*i.e.,* transformation) of the `maintainNbContainer` increments the number of containers which may satisfy the precondition of the `checkForSpeed` (*i.e.,* the number of carried containers reaches a threshold number). In addition, the graph shows that the `checkForTemperature` advice of the `craneSafety` inhibits the execution of the `maintainNbContainer` advice of the `saveEnergy` aspect. That is because, the `checkForTemperature` advice may skip the execution of the intercepted join point (*i.e.,* `load` and `unload` method calls) which prevents the execution of the `maintainNbContainer` advice.

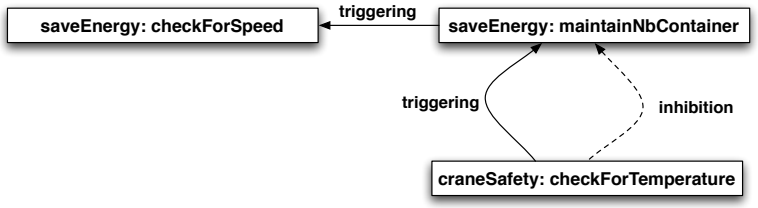

Figure 2.4: Weaving Interaction graph of the crane example showing the interaction between the `saveEnergy` and the `craneSafety` aspects

Conflicts are proven to be inhibition cycles of length greater than one in the graph. For some cases, an automatic resolution is made by an algorithm that diagnosis the graph and associates a weaving order to the transformations (advices). In other cases, users are asked to give more specific information about the application of CTs or to specify directly the right execution order for conflicting transformations. The main drawback of the approach is its focus on interferences that appear at the weaving stage. However, several other kinds of interferences cannot be detected within the approach. In our opinion, this is due to the limitation of the considered interactions between advices. Triggering and inhibition interactions are not enough. Take for example, the `saveEnergy` and the `performance` aspects; they are in conflict because the effects and the postconditions of their advices are contradictory (one wants the crane to move fast and the other wants it to move slow). This does not appear as a cycle in a graph based only on inhibition and triggering of advices.

2.2.2.2 Non-Modular Approaches

De Fraine et al. [Fraine 2008] propose a static control flow analysis of aspect interactions. Within control flow analysis, aspect interferences on non shared join points can be detected. The approach associates a set of *policies*. A policy is a predicate formulae that specifies the expected aspect-aspect or aspect-base control flow relations. A set of interesting predefined predicates are presented to define the relationships among aspects and base programs. For example, $must(a.m_1, b.m_2)$ predicate specifies that the aspect advice method $a.m_1$ occurs on all the control-flow of the base method $b.m_2$; $depend(b.m, a_1.m_1, a_2.m_2)$ predicate specifies that the advice methods $a_1.m_1$ and $a_2.m_2$ occur conjointly in the control-flow of the base method $b.m$ and $exclude(b.m, a_1.m_1, a_2.m_2)$ specifies that the two advice methods $a_1.m_1$, $a_2.m_2$ cannot occur together in the same control-flow of the base method $b.m$.

Control-flow graphs are automatically generated form the static analysis of bytecode programs with woven aspects. These graphs are traversed to define for each method the different paths traversed from its entry to its exit. Each path is defined as the set of methods (*i.e.,* nodes) visited. These paths are checked later to evaluate the different predicates. For example, $must(a.m_1, b.m_2)$ is defined as: $\forall p : path(b.m_2, p) \Rightarrow member(a.m_1, p)$ which states that for each path on the control-flow of $b.m_2$ the advice method $a.m_1$ is a member of that path. Here we show the policy associated to the `performance` aspect of the crane example.

$\forall M_1, M_2, M_3, A : matches(M_1,$ `Engine.move*(..)`$)\wedge$
$matches(M_3,$ `Performance.changeSpeed()`$)\wedge$
$adviceOf(A, M_2) \Rightarrow exclude(M_1, M_2, M_3,)$

Listing 2.7: The `performance` aspect policy

The above policy states that, in the control-flow of any method matching the pattern `Engine.move*(..)`, there is no other aspect advice (`adviceOf(A,` M_2`)`) that occurs conjointly with the `changeSpeed()` advice of the `performance` aspect. This latter policy is violated because the `saveEnergy` aspect executes its `checkForSpeed` advice on the `moveUp` and `moveDown` methods of the `Arm` that belong to the control-flow of `Engine.moveLeft(Speed)` and `Engine.moveRight(Speed)` and hence an interference is detected. However, this is not completely true because not all the aspect advices should be excluded in this case, we need only to exclude those changing the parameter value of type `Speed`. In order to solve this problem, the policy language should be extended with data-flow analysis support.

Weston et al. [Weston 2007] developed AIDA tool (Aspect Interaction Detection Analysis). AIDA extends AspectJ compiler (*abc*) with data-flow analysis. Within AIDA a conflict is detected when a data variable used by one aspect advice is modified by another advice. It is the case for the `performance` and the `saveEnergy` aspects where their advices affect the `speed` parameter value of basic method calls. Besides the direct interaction, AIDA considers transitive interaction between as-

pects. This expresses the case when a data variable modified by one aspect is passed through a chain of base program methods before it is used by another aspect. AIDA performs a pre and post-weaving analysis. At pre-weaving stage, the control flow graph (a subgraph for each method considering its statements and the relationships with other methods) is analyzed and a *transfer function* and a *summary transfer function* are calculated. The *transfer function* indicates the effects of each statement on data variables. The summary transfer function is the conjunction of the *transfer function*s of all the statements of a method, in other words, it indicates the effects of calling a method. At post-weaving stage (when advices are injected into base program methods), the *summary transfer function*s of methods are recalculated and compared with those resulting from the pre-weaving stage to extract those of advice methods. The result information about data variables affected by the different advices are compared to detect potential direct or transitive data conflicts.

Katz et Sihman [Katz 2004] propose the use of Bandera tool [Hatcliff 2001] to model check a collection of aspects called *superimposition* and their interferences when they are applied to a base program. Bandera tool analyses Java source code augmented with aspects and annotated with BSL (Bandera's temporal Specification Language) formulas and generates a compatible code that can be checked using model checkers such as: SPIN [Ben-Ari 2008] and Java PathFinder [Havelund 2000]. The proposed verification process by the authors has four main steps: first the base system should be correct with respect to a set of basic properties defined for this purpose. Second, a set of assumptions defined for each aspect in the superimposition should be checked, thus when they are not satisfied the aspects cannot be woven into that base system. Third, weave the superimposition to the base program and check whether the basic properties used in the first step are still verified, otherwise an interference with the base program is reported by returning a counter example. Fourth and finally, a set of desired properties of the woven aspects are checked. The idea behind the proposition of the superimposition concept is to provide a reusable aspect library. That is to say, when a superimposition is woven into a basic program satisfying the superimposition's assumptions, it is assured that the woven program will satisfy the desired properties and conform to the original specification of the basic program.

In a recent work, Chen et al. [Chen 2010] propose a formal model based upon UTP (Unified Theories of Programming) designs [Hoare 1998] for base-aspect and aspect-aspect interferences. In this work, base classes and aspects are defined as tuples of their underlying elements (*i.e.,* attributes, methods, advices, intertype declarations). Methods and advices are defined as UTP designs of the form $p \vdash q$ that can be read as: if the execution of a design starts successfully from a state in which the precondition p holds, it will terminate successfully in a state satisfying the postcondition q. The pre and postconditions are defined in terms of class attributes values. Moreover, an invariant is defined for each class. An invariant is a global predicate that is assumed to be hold before and after the execution of each method of the class. The weaving is modeled as the composition of the formal models describing

the class and the aspects to be woven. An interference between an aspect and a base program is detected by checking whether the base program design satisfies the aspect preconditions. That is to say, if the aspect advice is a before advice, the result predicate of the conjunction of the class invariant and the precondition of the method to be altered by the aspect should imply the precondition of the advice. Similar rules are defined for the after and around advices. After weaving, the result method should satisfy the invariant of the class, otherwise an interference is reported. An interference between two aspects is detected by: first check the potential weaving of the aspects individually to the base program; second, weave the first aspect and check weavability of the second aspect with the woven system, finally, check the postcondition of the first aspect whether it is still satisfied after the weaving of the second aspect. If it is the case, the two aspects are interference-free otherwise, an interference is reported. In this case, the unsatisfied conditions can be used to avoid the conflict. The main drawback of this approach, is that aspects are applied to a single class and hence aspects altering several classes cannot be treated. Table 2.1 summarizes the above discussed approaches for aspect interferences detection and resolution, the underlying logic and the verification tool used by each approach.

2.3 Lessons learned

In our assessment process we focus on interferences detection and resolution in general. In the following we list the set of conclusions from the above review over interferences detection and resolution:

2.3.1 Interference Detection

1. Syntactic verification is one first step towards interferences detection. Nowadays, it is known that these approaches are not efficient enough to detect all potential interferences among aspects, thus semantic interference verification should be considered. This includes data and controlflow analysis. This need can be justified by the fail of syntactic-based approaches to detect the conflict between the `performance` and the `saveEnergy` aspects of our crane example.

2. As shown in [Fraine 2008, Weston 2007], an efficient semantic interference detection must include both control and data flow analysis. This enables a precise and a complete analysis of aspect advices interactions.

3. An aspect specification should be unaware of any other aspect, otherwise a new specification is needed each time has an aspect to be woven to another context. The compositional intentions based specification provided by Marot et al. [Marot 2009] is a good example implementing this purpose, where CTs are defined in an abstract way and do not rely on a specific context.

As general conclusion for aspect interferences detection, both syntactic and semantic features of aspects must be considered. In other words, an effective detection

of aspect interferences must take into account the aspect *intent* and *behavior*. While the intent of an aspect explains the aspect effects on the base system, the behavior shows the positions where the aspect interact with the system and the explicit actions taken by the aspect to achieve the intent. In addition, the specification of intents and behaviors of aspects should be as abstract as possible for testing the applicability of an aspect to any base system and check its interaction with any aspect applied to such a system.

2.3.2 Interference Resolution

1. Precedence is the basic and intuitive way to compose aspects but it does not solve all kinds of interferences. Thus, we should get rid of aspect precedence study and think about more powerful composition strategies.

2. The compositions strategies should be generic and not ad hoc propositions, this enables the reusability of the proposed solutions.

3. AOP approaches generally use an implicit specification of skipping a join point. This needs to be revised for interference resolution, because the different aspects actions on join points should be considered. Take for example the `craneSafety` and the `saveEnergy` aspects: when `load` or `unload` calls are skipped by the former aspect, the corresponding advice of the latter aspect should not be executed otherwise the `saveEnegy` aspect will forces the `arm` to work in slow mode before reaching the threshold number of carried containers. This cannot be detected without making the skip action explicit. In [Marot 2009], skip is modeled by the `NoProceed` action to solve the conflict.

4. Generally, an interference can not be solved automatically. In this case it is not sufficient to report the conflict to the user, useful and aiding information should also be reported so that the user could realize the source of the conflict and can make the right decision. It is the case for the UTP based model provided by Chen et al. [Chen 2010].

As general conclusion for interference resolution, the actions of aspects on join points should explicitly be specified, and more general and reusable composition strategies must be provided. In addition, the developer should be given information that explains why and how an interference appears. This enables the developer to choose the right composition strategy that solves the interference. We believe that model checking approaches may help in the detection of interferences and give sufficient information about aspect properties violation. We also believe that a library of patterns for composition operators may help to determine the right composition strategies solving interferences.

Model reference	Interference Analysis		Underlying logic	Verification tool
	Detection	Resolution		
[Durr 2008]	data/control flow analysis	precedence	cflow graph	AVM
[Marot 2009]	compositional intentions on advices	precedence	dflow/ cflow graphs	-
[Douence 2002]	events observation	operators: Seq, And, Or	FSP	LTSA
[Katz 2008]	assume/guarantee proprieties	-	LTL	Maven
[Kniesel 2009]	cflow graphs	precedence	FOL	-
[Fraine 2008]	assume/guarantee properties	-	FOL	cflow analysis
[Weston 2007]	data flow analysis	-	dflow graphs	AIDA
[Katz 2004]	static code analysis	-	TL	Bandera, SPIN
[Chen 2010]	pre/post conditions	-	UTP	-

Table 2.1: Summary of AOP models and their supports for aspect interferences detection and resolution: (TL) Temporal Logic, (LTL) Linear Temporal Logic, (FOL) First Order Logic, (UTP) Unifying Theory of Programming, (AVM) Abstract Virtual Machine

Chapter 3

Component Based Software Engineering and their AOP support

Contents

3.1 Overview of CBSE

Component-Based Software Engineering (CBSE) enables quick and easy assembly of prefabricated software units named components to get the final software application. With CBSE, the required components are developed independently by software developers. Components come with a clear specification of what it provides and requires. That indicates how and with which a component can be assembled,

and under which circumstances the assembling is correct. The main benefits of CBSE is the modularity, the reusability and the interoperability of software components. By modularity, a software application is composed of separate components, each of which implements a part of the business logic of the system which deeply improves the maintainability of software applications. By reusability, components can be used again and again in different software applications with non-invasive adaptation, minimum reconfiguration or even with no modification at all. By interoperability, heterogeneous components that come from different providers can be assembled and work together efficiently. Nowadays, several industrial and academic component models are proposed (*e.g.,* CORBA, EJB, Fractal, Sofa, etc.) aiming to ensure better modularity, reusability and interoperability of components. However, as we mentioned in the above chapter, some functionalities or system features are non modular, and when they are needed to be implemented in CBSE, the big challenge is how to ensure the modularity of those functionalities and the reusability of regular components. One solution is to provide a support of aspect-oriented programming techniques to CBSE.

In this chapter we overview a set of component models and we discuss their support level of aspects. The overview does not mean to be exhaustive, but rather discuss their aspect-oriented support. For better comprehensibility of how the presented models work we demonstrate their ability to model our crane example with its different aspect extensions. Note that each component model, provides its own definitions of architectural elements and their relationships. Here, we give the common terminologies of the CBSE that we use all along in this chapter.

Component a component is the composition unit of CBSE, it is a black or gray box that encapsulates a business logic and exposes its services through a set of interfaces. For hierarchical component models, two kinds of components are supported: *primitives* and *composites*. Primitive components are the fine-grained software units of composition, while composites are units composed of other components either primitive or composite ones.

Interfaces interfaces are the only access points of a component. Services provided by a component are abstracted and exposed in interfaces named *provided interfaces*. Components that require services from other components abstract all the required services in separate interfaces called *required interfaces*.

Binding is a mechanism of assembling components together. A binding connects a required interface to one or more provided interfaces directly or by means of connectors.

The presented component models in this chapter are divided into three categories: *container-based* and *aspectual* component models, and *software architecture modeling-based* models.

3.2 Container-Based Component Models

Container-based component models are developed to free software developers from implementing the so called complex features or aspects in AOP such as distribution, synchronization and persistence. So that, the containers implement and manage these aspects for the deployed components. Enterprise Java Beans and CORBA component models belong to such category. The main drawbacks of such component models are: (1) restricted aspects provided by their containers (2) do not support flexible aspect extensions. However, some extensions are proposed to extend those models with user defined aspects. In this section, we discuss a set of those component models with their extensions.

3.2.1 EJB

EJB [Burke 2006] enables the development and the deployment of components which are Java objects called *enterprise beans*. Enterprise beans are distributed software components that run exclusively in an EJB container. The EJB container wraps enterprise beans and prevents any direct access to their provided services. Each external invocation to an enterprise bean service is intercepted by the container; the container executes some additional behaviors according to the semantic of its supported aspects then it may calls the original invoked service. The EJB container supports security, persistence, transactions, concurrency, and resources management aspects. These are predefined aspects that are automatically woven to the enterprise beans and do not have to be written by enterprise beans developers.

Each EJB component exposes two interfaces *home* and *remote*. While the former provides life cycle management services for the bean: creation, update, destruction, and locate the bean reference, the latter provides business services of the bean. For the crane example, `IController`, `IEngine`, `IArm`, and `IMagnet` interfaces are remote interfaces because they define business services of their components. Using a business service of the bean by an external enterprise bean is made by first calling the home server interface to get a reference to the bean then, the service of the remote interface can be invoked. Figure 3.1 shows the structure of the EJB container. EJB distinguishes two kinds of components: *entity* and *session* beans. Entity beans represent a data in a database and their remote interface only define accessors and other related services to that data. Session beans implement business services of their remote interface. In our crane example, all the components are session beans because they implement business services that are not related to any database. The main advantage of using EJB is the portability of enterprise beans. By portability, once an enterprise bean is developed, it can be executed in any EJB container such as the ones provided by BEA, IBM, and GemStone servers.

From the AOP point of view, pointcuts of the supported aspects by EJB are explicitly indicated in an XML file called *deployment descriptor*. In this file, enterprise beans developers explicitly specify the methods that require a transaction, or a security access role. All these information and others like whether the persistence

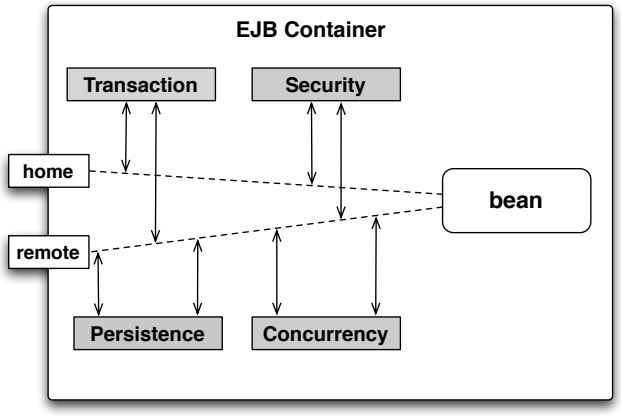

Figure 3.1: EJB Container structure

is handled automatically by the container or is performed by the bean itself is collected in the deployment descriptor. The file is read by the EJB container at the deployment phase to know how to manage the beans at runtime. Listing 3.1 shows the deployment descriptor file of the crane example. In the listing, each component is defined using `session` element (for session beans only) (line 3-8), everyone has a full access to all the component methods (line 12-21) with the use of `"*"`, and transaction is not required for methods with the use of `Never` transaction type (line 27).

Limitations

At first glance, it appears that aspect weaving in EJB is made by wrapping, but in fact, the enterprise beans must implement the `EJBHome` and `EJBObject` interfaces. These interfaces define methods to be implemented by enterprise beans to explicitly call the container to execute the code of the predefined aspects for them. The main drawback of EJB, is its lack of support for user defined aspects such as the `performance` and the `saveEnergy` aspects required for our example. Furthermore, EJB does not provide a flexible way to update and compose the current predefined aspects.

3.2.2 AES

AES [Choi 2000] extends EJB with a development support of flexible and extensible aspects. It replaces the default EJB container with *metaobjects*. With AES, aspects are independent entities that are developed and updated separately then added to metaobjects. A metaobject controls a *baseobject* (*i.e.,* regular enterprise bean) and references one or more aspects and calls them when method calls, to the baseobject, are intercepted.

The deployment descriptor of EJB is extended to cope with user defined aspect pointcuts. In the new deployment descriptor one should indicate the set of methods

```
1 <ejb-jar>
2   <enterprise-beans>
3     <session>
4       <ejb-name>ControllerBean</ejb-name>
5       <home>ControllerHome</remote>
6       <remote>IConcoller</remote>
7       <ejb-class>Controller</ejb-class>
8     </session>
9     <! all the other components are declared here the same way/>
10  </enterprise-beans>
11  <assembly-descriptor>
12    <security-role>
13      <role-name>everyone</role-name>
14    </security-role>
15    <method-permission>
16      <role-name>everyone</role-name>
17      <method>
18        <ejb-name>*<ejb-name>
19        </method-name>*</method-name>
20      </method>
21    </method-permission>
22    <container-transaction>
23      <method>
24        <ejb-name>*<ejb-name>
25        </method-name>*</method-name>
26      </method>
27      <trans-attribute>Never</trans-attribute>
28    </container-transaction>
29  </assembly-descriptor>
30 </ejb-jar>
```

Listing 3.1: Deployment descriptor file for the crane system in EJB

affected by each aspect. The deployment descriptor is read by the metaobject to call the different aspects when join points are captured at runtime. Listing 3.2 illustrates an excerpt of the deployment descriptor of the crane example. Line 1 states that all the methods (*i.e.,* `"*"`) of the `Controller` bean are connected to the `setLoadingState` advice method of the `truckSafety` aspect. Line 2 states that the method `moveDown` of the `Engine` is connected to the `checkForSpeed` advice method of the `truckSafety` aspect.

```
1 Controller is truckSafety with {setLoadingState = *}
2 Engine is truckSafety with {checkForSpeed = moveDown}
3 Controller is saveEnergy with {maintainNbController = *}
4 Engine is saveEngine with {checkForSpeed = *}
```

Listing 3.2: deployment descriptor in AES

In the case of shared join points, aspects advice methods are called in nested way. Listing 3.3 illustrates a method called `nested_load` of the metaobject. This latter, connects the `Controller` baseobject with the `saveEnergy` and the `truckSafety` aspects for the `load` method that should be intercepted by both aspects. A nested method `m` is created (line 5) for the `load` method of the baseobject `Controller`. The `truckSafety` and the `saveEnergy` aspects with their corresponding advice methods

(*i.e.,* `setLoadingState` and `maintainNbContainer` respectively) are added to the nested method in such order (line 6-7). Finally, the nested method is invoked (line 8). Following code nesting principle [Choi 2000], the created nested method executes the last added aspect advice, then the one before the last and so forth until the first added one, all around the base method.

```
1 Object nested_load() {
2   Method base = controller.getClass().getMethod("load");
3   Method ad1 = truckSafety.getClass().getMethod("setLoadingState");
4   Method ad2 = saveEnergy.getClass().getMethod("maintainNbContainer");
5   NestedMethod m = new NestedMethod(this, controller, base);
6   m.addMethod(truckSafety, ad1);
7   m.addMethod(saveEnergy, ad2);
8   return m.invoke() ;
9 }
```

Listing 3.3: `truckSafety` and `saveEnergy` composition in AES

Limitations

AES extends EJB with a support for user defined aspects. However, nesting composition mechanism is just another form of sequential composition of aspects. It correctly compose the `truckSafety` and the `saveEnergy` aspects but it is useless for aspects not sharing join points such as the `saveEnergy` and the `performance` aspects. Thus, much work is still needed to support more powerful composition mechanisms. In addition, AES does not provide any support for aspect interferences detection.

3.2.3 CORBA/CCM

CCM [Wang 2001, OMG 2006] extends CORBA with a development support to implement, configure, and manage components. In CCM, component interfaces are called *ports*, the set of provided interfaces are called *facets*, and the set of required interfaces are called *receptacles*. A component in CORBA/CCM can emit and push events via two more kinds of ports called *event sources* and *event sinks*, respectively. In other words, facets and receptacles are used for synchronous communications, while event sources and sinks are used for asynchronous communications. Moreover, CCM defines component instance managers called *homes*; a home is defined for each component type and manages the life cycle of its component instances. Like EJB, CORBA components are managed by containers; a container encapsulates components and component homes and forwards client requests to components it manages to the *Object Request Broker* (ORB) to execute some predefined aspects: transaction, security, persistence, and notification services. Furthermore, a container defines a set of API interfaces accessible by its inner components to explicitly call the ORB aspects via their internal interfaces. Figure 3.2 shows the CCM container model.

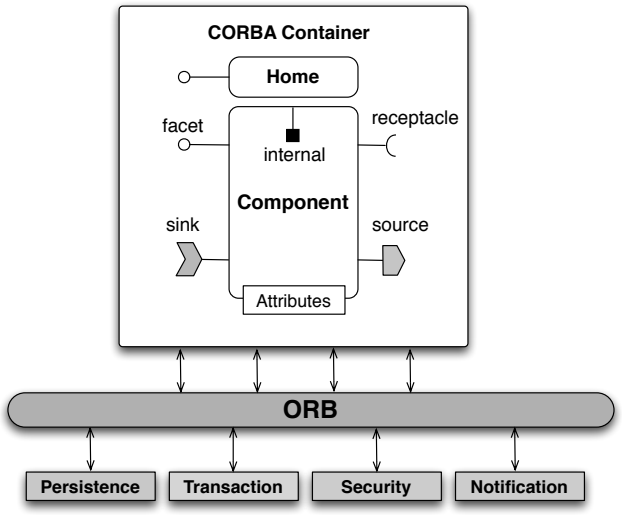

Figure 3.2: CORBA Container model

Similar to EJB, CCM provides XML descriptors called IDLs. CCM descriptors are used to describe components, properties, assemblies, and packages. Component descriptors are used to describe component features: name, ports and home. Property descriptors are used to configure component instances at the deployment phase by assigning values to component attributes. Assembly descriptors are used to describe system configurations or component connections. Finally, package descriptors are used to define a set of general technical properties of components such as the supported operating system, the implementation language, and the component dependencies (*i.e.,* the set of required programs for the execution of the component). Listing 3.4 describes the `Controller` component in IDL component descriptor; provided interfaces are noted with `provides` (line 2), required interfaces are noted with `uses` (line 3), and the component home `ControllerHome` is explicitly defined with the keyword `home` (line 5).

```
1 component Controller {
2   provides IController iController;
3   uses IEngine iEngine;
4 }
5 home ControllerHome manages Controller{}
```

Listing 3.4: `Controller` component in CCM/CORBA IDL

Limitations

CORBA is a language and platform independent model and its *Portable Object Adapter* (POA) extension [Pyarali 1998] makes CORBA containers more flexible than those of EJB, but not flexible enough to allow the definition of user defined aspects.

3.2.4 AspectCCM/CORBA

AspectCCM [Clemente 2002] extends CCM component model of CORBA with a support for user defined aspects. Aspects in AspectCCM are regular CCM components exposing their services via facets and event sinks. AspectCCM distinguishes two kinds of component interconnections: *intrinsic* and *non-intrinsic* interconnections. The former indicates the business logic services required from other CCM components, these are normally specified in CCM with *uses* keyword in the component specification phase (see Listing 3.4). The latter indicates non business logic services implemented by aspects; these required services are added later at the assembly or the packaging phases. AspectCCM adds *uses_aspect* keyword to the component specification to describe this second type of interconnections. Now, the `truckSafety` and the `saveEnergy` aspects can be added to the `Controller` component description as shown in Listing 3.5 (line 4-5). Note that the `truckSafety` and the `saveEnergy` aspects also require the interception of the `Engine` component interfaces and hence they should also be added to the `Engine` component descriptor.

```
1 component Controller {
2   provides IController iController;
3   uses IEngine iEngine;
4   uses_aspect truckSafety;
5   uses_aspect saveEnergy;
6 }
7 home ControllerHome manages Controller{}
```

Listing 3.5: `Controller` component in AspectCCM/CORBA

Aspect services are added at the packaging phase. For this purpose, AspectCCM extends the package software descriptor of CCM with elements defining the package where aspects are hosted, where and when each aspect advice is called in AspectJ like expressions. Listing 3.6 illustrates a simplified code of the IDL extension provided by AspectCCM to describe aspect weaving in our crane example. Pointcuts are defined in line (2-3) with `pointcut` element, while the aspect advices association is specified with `advice` element. This description is later parsed and dependencies of aspects are added to CCM components without introducing crosscutting to components implementation. In AspectCCM it is at the assembly phase where the real instances of aspects are bound to the base components. For this purpose, the assembly descriptor of CCM is also extended to support aspect connections as shown in Listing 3.7 where aspect instances are located (line 2-3).

Limitations

AspectCCM enables user defined aspects and aspect instantiation. However, it does not say anything about the execution order of aspects even at shared join points. In other words, AspectCCM does not provide a support for explicit composition of aspects. In addition, aspect interference detection is omitted by the model.

```
<usesaspects>
  <pointcut name="in" type_element= execution(* Controller.*())/>
  <pointcut name="out" type_element= call(* Engine.*())/>
  <advice nameserver="truckSafety" pointcut_name="in" type= before/>
  <advice nameserver="saveEnergy" pointcut_name="in" type=before/>
  <advice nameserver="truckSafety" pointcut_name="out" type= before/>
  <advice nameserver="saveEnergy" pointcut_name="out" type=before/>
</usesaspects>
```

Listing 3.6: Aspect weaving description in AspectCCM

```
<connections type=connectaspects>
  <aspect componentinstanceref= truckSafety/>
  <aspect componentinstanceref= saveEnergy/>
</connections>
```

Listing 3.7: Aspect instances locators in AspectCCM

3.2.5 Spring AOP

SpringAOP [Pawlak 2005, Walls 2007] is a container based component model that allows user defined aspects. Aspects in SpringAOP are Java objects called *beans* implementing the different advice methods. To be weaved to other beans implementing the business logic of applications, aspect pointcuts must be defined using either AspectJ annotation style or schema-based approach. In the former case, annotations defining pointcuts and advices are scattered over different beans. In the latter case, a separate file of an XML-like schema is used to define aspect elements. Here we focus on the latter case, Listing 3.8 shows the schema-based description of the `saveEnergy` aspect. Each aspect is defined by `aop:aspect` element with an identifier and a reference to the bean implementing it (line 2). Pointcuts are described by `aop:pointcut` element with an identifier and an AspectJ-like pointcut expression (line 3-4). Currently, SpringAOP supports only method execution pointcuts. Advices are described according to their types, in the `saveEnergy` case, only before advices are used, they are defined by `aop:before` with a pointcut reference `pointcut-ref` and the advice method `method` (line 5-6).

```
<aop:config>
  <aop:aspect id="saveEnergy" ref="saveEnergyBean">
    <aop:pointcut id="in" expression="execution(* Controller.*(..))"/>
    <aop:pointcut id="out" expression="execution(* Arm.*(..))"/>
    <aop:before pointcut-ref="in" method="maintainNbContainer"/>
    <aop:before pointcut-ref="out" method="checkForSpeed"/>
  </aop:aspect>
  <bean id="saveEnergyBean" class="saveEnergy"/>
</aop:config>
```

Listing 3.8: `saveEnergy` aspect in SpringAOP schema-based approach

Limitations

SpringAOP provides a flexible container for beans. It enables user defined aspects and interoperate with AspectJ weaver. Form AOP point of view, SpringAOP is a mapping of AspectJ concepts into Spring beans: it supports all the advice types defined in AspectJ (*i.e.,* before, after, around, before throwing, etc.), and supports aspect instantiation using AspectJ's like mechanism (*i.e.,* `perthis` and `pertarget`). For aspect weaving, SpringAOP may call AspectJ weaver at loading time to weave Spring beans. SpringAOP also supports weaving by wrapping beans with proxies that intercept calls and delegate them to aspects relevant to that method calls. The former approach is used when both internal and external calls needed to be intercepted, because with proxy-based approach only external calls are intercepted. Composing Aspects follows AspectJ precedence mechanism (the aspect beans should either implement the predefined `Ordered` interface or be annotated with `@order` notation). However, aspect interferences detection is also omitted by this model and the precedence relationship among aspects is not sufficient to solve interferences.

3.2.6 JBoss AOP

JBossAOP [Pawlak 2005] provides a support for AOP to Java component systems. Like SpringAOP, aspects are regular Java classes extended with annotations or with XML-descriptions to explicitly describe pointcuts and advices. The join point model of JBossAOP supports method and constructor execution, attribute access and control flow specification. Listing 3.9 shows a description of the `saveEnergy` aspect in XML-description. Pointcuts and advice associations are defined with `bind` element (line 3-5, 6-8). Different instances of aspects may appear at the same application, this can be specified in the `scope` attribute of the `aspect` element (line 2). `PER_VM` value indicates that only one aspect instance should appear for an application. Other values can be used to instantiate aspects for each class `PER_CLASS`, for each bean instance `PER_INSTANCE` and even for each join point (`PER_JOINPOINT`). JBossAOP also supports intertype declaration using mixins.

```
1 <aop>
2   <aspect name="saveEnergy" class="saveEnergy" scope="PER_VM"/>
3   <bind pointcut="execution(* Controller.*())">
4     <before name="maintainNbContainer" aspect="saveEnergy"/>
5   </bind>
6   <bind pointcut="execution(* Arm.*())">
7     <before name="checkForSpeed" aspect="saveEnergy"/>
8   </bind>
9 </aop>
```

Listing 3.9: `saveEnergy` aspect in JBossAOP schema-based approach

Listing 3.10 explicitly shows that the `setLoadingState` advice of the `truckSafety` should be executed before the `maintainNbContainer` advice of the `saveEnergy` aspect (line 2-3). The same, the `checkForSpeed` advice of the `truckSafety` should be executed before the `checkForSpeed` advice of the `saveEnergy` aspect (line 6-7).

```
1 <precedence>
2   <advice aspect="truckSafety" name="setLoadingState"/>
3   <advice aspect="saveEnergy" name="maintainNbContainer"/>
4 </precedence>
5 <precedence>
6   <advice aspect="truckSafety" name="checkForSpeed"/>
7   <advice aspect="saveEnergy" name="checkForSpeed"/>
8 </precedence>
```

Listing 3.10: Advice precedence in JBossAOP

Limitations

JBossAOP also maps AspectJ concepts into JBoss beans. JBossAOP allows dynamic weaving by wrapping beans (*i.e.,* JBoss intercepts join points at runtime and calls aspects). No interference detection mechanism is provided and only advices precedence at shared join points is supported.

3.2.7 JAsCo

JAsCo [Suvée 2003] extends Java beans with aspects. Aspects or *aspect beans* in JAsCo define a set of *hooks*. Each hook is defined by one or more constructors and one or more advice methods. A hook constructor initializes a hook; its body specifies an abstract pointcut over an abstract method parameter. When it is instantiated, the abstract method matches a concrete method and as a result, the abstract pointcut matches a concrete pointcut. In JAsCo, only two AspectJ-like pointcuts are available: *execution* and *cflow*. Like AspectJ, JAsCo supports *before*, *after* and *around* advices. Listing 3.11 shows the code of the `saveEnergy` aspect in JAsCo with two hooks definition: `maintainNbContainer` (line 5-12) and `checkForSpeed` (line 14-21). The former matches each method execution (line 7) and maintains the number of carried containers by executing the before advice (line 9-11). The latter matches each method execution (line 16) and checks whether the number of carried containers reaches a threshold number (line 19). If it is the case, it changes the speed parameter value to `LOW`.

JAsCo uses *connectors* to instantiate and apply aspects to one or more components. A connector instantiates one or more hooks by calling their corresponding constructors. A constructor is called with an expression defining the set of concrete methods to replace its abstract method parameter. Listing 3.12 shows the `saveEnergyConnector`. This connector, instantiates the `maintainNbContainer` hook on each method defined within the `Controller` component whose name ends with *load*, and instantiates the `checkForSpeed` hook on each outgoing event (`onevent` keyword) of the `Engine` whose name starts with *move*. Hence, whenever a *load* or *unload* methods are called, the before advice of the `maintainNbContainer` hook is executed, and whenever *moveLeft* or *moveRight* events are triggered, the before advice of the `checkSpeed` hook is executed. JAsCo applies aspects to components via component wrapping. Wrapping is made by associating traps to methods, so that, when

```
1 class saveEnergy {
2   final static int THRESHOLD = 100;
3   int nbContainer = 0;
4
5   hook maintainNbContainer {
6     maintainNbContainer(method(..args)) {
7       execute(method);
8     }
9     before() {
10      nbContainer++;
11    }
12  }
13
14  hook checkForSpeed {
15    checkForSpeed(method(..args)) {
16      execute(method);
17    }
18    before() {
19      if(nbContainer>=THRESHOLD) ((Speed)args[0])=Speed.LOW;
20    }
21  }
22 }
```

Listing 3.11: `saveEnergy` aspect in JAsCo

```
1 connector seveEnergyConnector {
2   saveEnergy.maintainNbContainer nbc =
3     new saveEnergy.maintainNbContainer(* Controller.*load(*));
4
5   saveEnergy.checkForSpeed cs =
6     new saveEnergy.checkForSpeed(onevent * Engine.move*(*));
7 }
```

Listing 3.12: `saveEnergyConnector` in JAsCo

a method is called, a connector registry is interrogated and the concerned connector is called. The connector dispatches the called method or event to the corresponding hook. When more than one hook correspond to the method, a composition strategy assigned to the connector is executed. JAsCo supports composition of both advices and aspects. The advices' execution order is explicitly specified at each connector, when its is omitted, the before advices are executed first, then the around advices then the after advices. Listing 3.13 shows the `saveEnergyTruckSafetyConnector`. This latter, instantiates the `checkForSpeed` hook of both aspects `saveEnergy` and `truckSafety`. The execution order specification part (line 8-9) indicates that the before advice of the `truckSafety` aspect (line 8) is executed first then the before advice of the `saveEnergy` aspect can be executed (line 9).

Aspect composition in JAsCo is supported by implementing a `CombinationStrategy` interface that defines a `verifyCombinations` method to specify the execution strategy of aspects. The implementation of the combination strategy is instantiated and added to the connector using `addCombinationStrategy(strategy)` method. Listing 3.14 describes a *Twin* combination strategy of two hooks `hookA` and `hookB`. This combination-strategy specifies that `hookB` should be removed whenever `hookA`

```
1 connector seveEnergyTruckSafetyConnector {
2   saveEnergy.checkForSpeed se_cs =
3     new saveEnergy.checkForSpeed(onevent * Engine.move*(*));
4
5   truchSafety.checkForSpeed ts_cs =
6     new truckSafty.checkForSpeed(onevent * Engine.moveDown(*));
7
8   ts_cs.before();
9   se_cs.before();
10 }
```

Listing 3.13: Ordering `saveEnergy` and `truckSafety` aspects in JAsCo

is not found (line 11-16). This way, the behavior of `hookB` is never executed, if the behavior of `hookA` is not performed.

```
1 class twinCombinationStrategy implements CombinationStrategy {
2   private Object hookA, hookB;
3
4   twinCombinationStrategy(Object a, Object b) {
5     hookA =a;
6     hookB =b;
7   }
8   HookList verifyCombinations(HookList hlist) {
9     if (!hlist.contains(hookA)) hlist.remove(hookB);
10    return hlist
11  }
12 }
```

Listing 3.14: The twin combination stratgy in JAsCo

JAsCo offers a set of keywords that enables creating multiple aspect instances. For instance, `perobject` creates a unique hook for every target object instance, while `perclass` creates a unique hook for every target class.

Limitations

JAsCo provides a set of interesting AOP features: aspectualize components by component wrapping, aspect instantiation, advices execution ordering rules and a programmatic way for aspect composition. However, the model is a language dependent and does not provide a support for aspect interferences detection.

3.3 Aspectual Component-Based Models

Aspectual component-based models are more flexible and extensible models compared with container-based ones. They enable user defined aspects definition and provides a greater support for their extension when more requirements are needed. In this section we overview two component models of this category: CAM/DAOP and Fractal component model with its different extensions.

```
1 <aspect role="SAVEENERGY">
2   <evaluatedInterface>
3     <method name="load"></method>
4     <method name="unload"></method>
5     <method name="moveUp">Speed</method>
6     <method name="moveDown">Speed</method>
7   </evaluatedInterface>
8   <implementation>
9     <name>saveEnergy</name>
10    <language>java</language>
11    <class>saveEnergy.class</class>
12  </implementation>
13 </aspect>
```

Listing 3.15: `saveEnergy` aspect in CAM/DAOP ADL

3.3.1 CAM/DAOP

CAM/DAOP [Pinto 2003, Pinto 2005] provides a component aspect model (CAM) and a dynamic execution platform (DAOP). CAM is a flat component model with support of aspects. Aspects in CAM are special kind of components that implement an *evaluated interface*. The evaluated interface defines an *eval(Message m)* method that implements the advices code of the aspect, and it is called whenever a message is intercepted by the DAOP platform. Role names are assigned to both components and aspects. Roles are used to reference architectural elements; the main advantage of using roles is the ability to replace aspects and components by others implementing the same roles without need to recompile the system. CAM/DAOP provides an ADL language to describe the architecture of the system with aspects. Listing 3.15 describes the `saveEnergy` aspect in CAM/DAOP ADL. In the listing, the `saveEnergy` aspect is defined within the `SAVEENERGY` role (line 1), the methods of interest with their parameters are listed in the `evaluatedInterface` section (line 2-7). The aspect real name, its programming language and its implementation class file are defined in the `implementation` section (line 8-12).

The join point model of CAM includes component creation and destruction and received and sent messages or events. Pointcuts and advice types are both defined by means of composition rules. Each composition rule defines the source and the target components of the interesting methods, the set of aspects to be weaved, and when aspects must be executed. Listing 3.16 describes the composition rule of the `saveEnergy` and the `truckSafety` aspects of the crane example. The source component roles (*i.e.,* USER and ENGINE) are declared at line 4, while the target component roles (*i.e.,* CONTROLLER and ARM) are declared at line 5. The messages of interest (*i.e., load*, *unload*, *moveUp* and *moveDown*) are explicitly defined at line 6. The aspects to be weaved are defined at line (8-9) and the type of advices used in this case is BEFORE_RECEIVE (line 6). In CAM/DAOP an aspect can be executed before receiving and sending messages or events (BEFORE_RECEIVE, BEFORE_SEND), after receiving or sending messages or events (AFTER_RECEIVE, AFTER_SEND), and before or after creation or destruction of components (BE-

```
1 <compositionRules>
2   <aspectCompositionRule>
3     <aspectRule>
4       <sourceComponentRole>USER ENGINE</sourceComponentRole>
5       <targetComponentRole>CONTROLLER ARM</targetComponentRole>
6       <BEFORE_RECEIVE>
7         <message>load unload moveUp moveDown</message>
8         <aspectList>truckSafety</aspectList>
9         <aspectList>saveEnergy</aspectList>
10      </BEFORE_RECEIVE>
11    </aspectRule>
12  </aspectCompositionRule>
13 </compositionRules>
```

Listing 3.16: `saveEnergy` and `truckSafety` aspect weaving in CAM/DAOP

FORE_NEW, AFTER_NEW, BEFORE_DESTROY, AFTER_DESTROY).

Aspects in CAM/DAOP can only be executed sequentially or concurrently: aspects declared in the same `aspectList` element of the composition rule are executed concurrently and those declared in different `aspectList` elements (*e.g.,* line (8-9)) are executed sequentially. DAOP platform applies the weaving at runtime. When several aspects are sequentially composed and a message of interest is intercepted by DAOP, the *eval()* method of the aspects are called sequentially; the aspects in this case forms a chain, thus if an error occurs, the *eval()* methods of the subsequent aspects in the chain are not called and the message never reaches its target.

Limitations

CAM/DAOP is designed to be a language-independent model, but currently it is only implemented in Java. However, the model has several limitations: (1) the model does not provide a declarative pointcut language, instead all the messages or events to be intercepted are explicitly declared one by one, (2) it only support sequential composition of aspects, (3) it does not provide advice execution ordering, and (4) it does not provide a support for interferences detection.

3.3.2 Fractal

Fractal [Bruneton 2004, Coupaye 2006] is an extensible, flexible and hierarchical component model developed by OW2 consortium. Besides its component hierarchization support, Fractal supports a set of interesting features such as: component sharing and aspects through component controllers. Each component in Fractal has a *content* and a *membrane*. The content encapsulates the business logic of a component, while the membrane exposes the provided and the required interfaces of a component and encapsulates a set of controllers implementing aspects applied to the component. Figure 3.3 shows the general structure of a component in Fractal with a content and membrane. By hierarchization, both *primitive* and *composite* components are supported by Fractal. Primitive components are distinguished from composites by the definition of their contents. The content of a primitive component

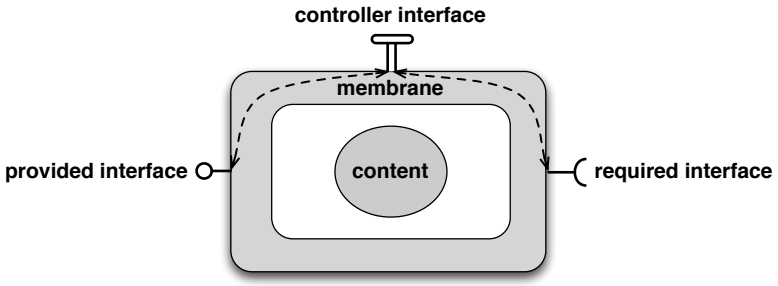

Figure 3.3: Fractal component architecture

is defined as a set of attributes and operations implementing the business logic of the component, while the content of a composite is defined as a set of components and their connections. Components are connected to each other via *binding*. A binding connects a required interface of a component to a provided interface on the assumption that the provided interface type is a sub-type of the required interface type due to the strong typing of Fractal.

Fractal also provides an architectural description language called Fractal-ADL to describe the architecture of component systems. Listing 3.17 illustrates the description of the `Crane` composite component in Fractal-ADL. The role attribute of an interface element indicates whether the interface is provided (*i.e.,* server) or required (*i.e.,* client). Primitive components content element indicates the location of the implementation class of the component, while the content of a composite component is defined by a set of component definitions encapsulated by the composite. Component binding is described using the binding element (line 14-16), this includes internal bindings that bound a composite component interface to an interface of its inner components. Internal bindings are indicated with the keyword `this` in the place of the component interface owner name (line 14, 16).

```
1 <definition name="Crane">
2   <interface name="iEngine" role="server" signature="crane.IEngine"/>
3   <interface name="iMagnet" role="client" signature="crane.IMagnet"/>
4   <component name="Engine">
5       <interface name="iEngine" role="server" signature="crane.IEngine"/>
6       <interface name="iArm" role="client" signature="crane.IArm"/>
7       <content class="cane.engineImpl"/>
8   </component>
9   <component name="Arm">
10      <interface name="iArm" role="server" signature="crane.IArm"/>
11      <interface name="iMagnet" role="client" signature="crane.IMagnet"/>
12      <content class="cane.armImpl"/>
13  </component>
14  <binding client="this.iEngine" server="Engine.iEngine"/>
15  <binding client="Engine.iArm" server="Arm.iArm"/>
16  <binding client="Arm.iMagnet" server="this.iMagnet"/>
17  <controller desc="Performance"/>
18 </definition>
```

Listing 3.17: The `Crane` component in Fractal-ADL

```
1 (Performance
2  (
3   'interface-class-generator
4   (
5     (performance-controller crane.IAspect)
6    )
7    (
8     (crane.Performance)
9    )
10   (
11    (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
12     org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
13     (FcInterceptors.inOutInterceptor 'interfaceName)
14    )
15  )
16  org.objectweb.fractal.julia.asm.MergeClassGenerator
17 'optimizationLevel
18 )
19 )
```

Listing 3.18: The specification of the `Performance` controller in Julia

Fractal supports dynamic reconfiguration through its provided API or by means of FScript language extension [David 2009a]. Furthermore, Fractal is a language-independent model and Julia [Bruneton 2006] is its reference implementation in Java. Fractal defines a set of predefined controllers to manage the life cycle of a component (life-cycle controller), to configure component attributes (attribute controller), to reconfigure the content of a composite controller (content controller) and to manage component interface bindings (binding controller). In addition, Fractal supports user defined controllers to implement aspects. Fractal controllers modify the behavior of their underlying components. A controller intercepts the messages sent and received by a component and it can alter them, redirect or even discard them. In Fractal-ADL aspects like controllers are defined using the controller element and the `desc` attribute indicates the name of a Julia template that describes the interface and the implementation class of the controller; this is specified in a separate Julia configuration file. For example, the `Performance` aspect of the crane example can be integrated as a controller to the `Crane` composite component (line 17 of listing 3.17) and the Julia configuration file is shown in Listing 3.18 where the interface controller (line 5), the class implementation file (line 8) and the type of interceptor needed (line 13) are specified.

Limitations

Fractal supports a set of CBSE features: component hierarchization, component instantiation and component sharing, in addition to its AOP support via the membrane controllers. Unfortunately, its AOP support is very limited: (1) no declarative pointcut language is provided, (2) no controller composition mechanism is offered, (3) controllers model only aspects altering single components (*i.e.,* primitive or composite), while aspects affect several components. Thus when the components involved in the interaction with an aspect do not belong to the same composite, a

simple controller cannot model the complete behavior of the aspect. For example, the `saveEnergy` concern affects the `Controller` and the `Engine` components. This aspect cannot be added to the system as a regular controller since the concerned components do not belong to the same composite. Several solutions are proposed; in the following we review the proposed solutions one by one highlighting their strengths and their weaknesses.

3.3.3 Fractal-AOP

Fractal-AOP [Fakih 2004] extends the membrane of components with two additional control interfaces: *execution controller* (noted by `cEc`) and *proceed controller* (noted by `sPc`). The former exposes the set of captured join points, while the latter proceeds any intercepted join point. The join point model of Fractal-AOP includes incoming and outgoing calls, getting and setting attributes, and components reconfiguration actions. An aspect in Fractal-AOP is modeled as an assembly of components: *advice components*, and a *weaving component*. Advice components are Fractal regular components that implement advices behaviors in services and expose them in one or more provided interfaces. The weaving component is responsible to decide what to do for each intercepted join point: proceeds it or call a service in an advice component. A weaving component must be configured to define the set of pointcuts to match join points at runtime and the services in the advice component to be invoked before and/or after join points execution. It proceeds the captured join point by calling the proceed controller of the functional component that advises the interception of that join point.

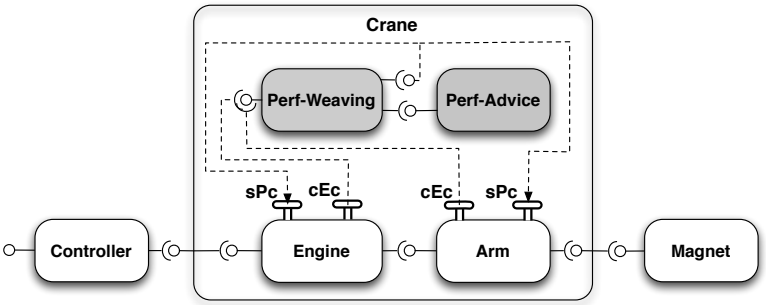

Figure 3.4: Fractal-AOP weaving of the `Performance` aspect to the crane system

Aspect weaving in Fractal-AOP is performed at runtime by binding the weaving component with both advice and functional components. For better understanding of Fractal-AOP principle, let us see how to model the `performance` aspect of the crane example in Fractal-AOP. The `performance` aspect is modeled as two components: `perf-Advice` and `perf-Weaving`. The former implements *setMagnetState* and *checkeForSpeed* methods and exposes them in a provided interface. The latter, is configured so that each captured call to *setOn* and *setOff* involves two actions: first, calling the *setMagentState* method on the provided interface of the

`perf-Advice` component, second, proceeds the call by calling the proceed controller of the `Arm` component. In addition, each captured call to *moveLeft* and *moveRight* involves: first, calling the *checkForSpeed* method on the provided interface of the `perf-Advice` component, second, proceeds the call by calling the proceed controller of the `Engine`. Figure 3.4 depicts the crane system architecture after weaving the `performance` aspect, the result bindings of the weaving process are drawn with dashed lines.

Limitations

Fractal-AOP provides, for Fractal, an API for explicit support of AOP. An aspect is separated into a weaving and advice components. This enables the reusability of aspects but complicates the architecture of the system by adding new components modeling aspects. Fractal-AOP does not provide a declarative pointcut language and does not say much about aspect composition and misses a support for aspect interferences analysis.

3.3.4 FAC

FAC [Pessemier 2006, Pessemier 2008] models an aspect as a regular component called *aspect component*. Like Fractal-AOP, an aspect component embodies a cross-cutting concern by implementing the different advices code as services and expose them in a provided interface called *advice interface*. The weaving component of Fractal-AOP is replaced by a controller called *weaving interface*. This latter implements a set of methods like *weave* that enables *aspect binding*. An aspect binding is a kind of connection of a controller interface to a regular provided interface, which is in fact a simple access call to aspect component interfaces via Fractal introspection. The weaving interface intercepts calls from component membrane and delegates them to the aspect component by calling its advice interfaces. In order to be able to weave different components at different levels of architecture, an *aspect domain* is proposed. An aspect domain is a composite that encapsulates the aspect component and shares the other underling components with their original components taking advantage of Fractal component sharing feature. Figure 3.5 illustrates the crane system after weaving the `performance` and the `saveEnergy` aspects following FAC approach. In the figure, aspect components are annotated with `«AC»` stereotype, shared components are represented by dashed boxes, aspect bindings are depicted with dashed lines, and aspect domains are indicated by gray boxes.

In order to define the set of components and their interfaces captured by an aspect, a pointcut language is provided. In addition to pattern matching, the pointcut language of FAC defines two keywords `SERVER` and `CLIENT` to indicate the provided and required interfaces by a component, respectively. For example: `SERVER controller.*.*` means that the aspect controller should intercept every method on every provided interface of the `Controller` component. The pointcut language of FAC also supports tracematch mechanism [Allan 2005, Avgustinov 2006]; trace-

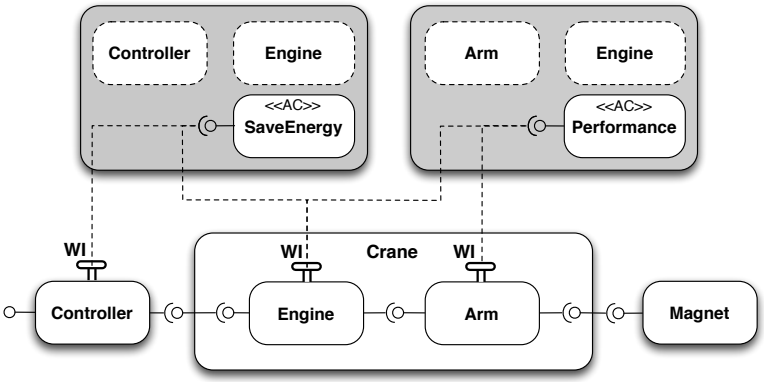

Figure 3.5: FAC implementation of the crane system

matching means that an aspect is executed only when a sequence of methods is executed. Regular expressions are used to define traces.

For shared join points, FAC aspect controller interface defines methods that assign an execution order to aspects at each common method [Pessemier 2007]: *changeACorder*, *getACPosition*. The former changes the execution order of an aspect component in a common method, the latter returns the current position of an aspect for a common method.

Limitations

FAC is a first step towards an expressive, declarative, and proper pointcut language for component models. Regular expressions defining traces and `SERVER` and `CLIENT` keywords are useful pointcut expressions. However, more expressive language is still needed for component models: with `SERVER` and `CLIENT` expressions, one should explicitly specify the component(s) whose interfaces needed to be intercepted, and pattern matching is not sufficient to ensure the required expressiveness of the language. FAC only supports precedence relationship among aspects execution at shared join points and does not provide a mechanism for aspect interferences analysis.

3.3.5 Safran

Safran [David 2003] extends Fractal component model with dynamic adaptation support. Adaptation is considered as a separate concern or aspect that can be defined separately and woven to the base system when needed at runtime. For this purpose, Safran extends Fractal component membranes with a controller interface that encapsulates a set of adaptation policies and a set of runtime information called *context-awareness* about components. These information are used to decide which adaptation policy should be triggered. An adaptation policy includes changing component bindings, adding or removing meta level components that implements the adaptation strategy. When an adaptation policy is triggered and a meta level

component is added, the meta level component intercepts method calls instead of the original component, executes some additional behavior and possibly calls the original method on the adapted component. Figure 3.6 shows the application of an adapted policy to a component.

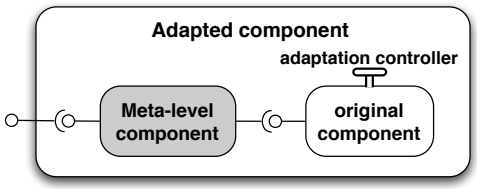

Figure 3.6: Safran Adaptation mechanism

Limitations

Safran suffers from the same limitations of Fractal-AOP, but the main drawback of the approach is that adaptation policies are applied only to individual components which is not compatible with aspects that affect more than one component such as the `performance` and the `saveEnergy` aspects of our crane example.

3.4 Software Architecture Modeling based models

The peculiarity of software architecture based models is that aspects are introduced at early phases (*i.e.,* analysis or design) of the modeling process. In this section we overview two approaches belonging to this category: PRISMA and AspectLEDA showing how aspects are defined, weaved and composed into regular components.

3.4.1 PRISMA

PRISMA approach [Pérez 2006, Pérez 2008] provides an architecture description language named AOADL to describe aspects, components (*i.e.,* primitives), connectors and systems (*i.e.,* composites) as architectural elements at the design stage. PRISMA follows the MDSOC approach [Tarr 1999, Ossher 2001] in considering both components and aspects as regular concerns. The main difference is that an aspect is the only concern that crosscuts different other concerns. An aspect is defined in a separate module that can be imported by several other architectural elements (*i.e.,* components and connectors). An aspect definition in PRISMA AOADL includes the definition of a set of attributes, a set of services, a set of roles an aspect may taken following the semantics of its defined services, and a protocol describing the aspect behavior. Listing 3.19 shows the `saveEnergy` aspect in PRISMA AOADL. The `saveEngery` aspect is considered to be an optimization concern that captures the `IController` and the `IArm` interfaces (line 1). It defines the `nbContainer` attribute (line 3), and the semantics it associates to each service (line 5-20). The `begin` and `end` are predefined services used for aspects creation and destruction,

respectively. The `in` keyword indicates that the service is offered by an aspect. The `played_roles` part indicates that the aspect plays two roles: maintain the number of carried containers for `load` and `unload` services (line 22), and speed checker for `moveUp` and `moveDown` services (line 23). The `saveEnergy` aspect protocol (line 25-26), says that an aspect after its creation until its destruction, plays its different roles in indeterministic way (line 24-26).

```
1 OPTIMIZATION aspect saveEnergy using IController, IArm
2   attributes
3     nbContainer : nat;
4   services
5     begin;
6     in load();
7       valuations
8         [in load()] nbContainer := nbContainer +1;
9     in unload();
10      valuations
11        [in unload()] nbContainer := nbContainer +1;
12    in moveUp(input speed:Speed)
13      valuations
14        [in moveUp(input speed:Speed)]
15          if (nbContainer>100) speed := Speed.LOW;
16    in moveDown(input speed:Speed)
17      valuations
18        [in moveDown(input speed:Speed)]
19          if (nbContainer>100) speed := Speed.LOW;
20    end;
21  played_roles
22    MAINTAIN_NB_CONTAINER ::= IController.load? () + IController.unload? ();
23    CHECKSPEED ::= IArm.moveUp? (Speed) + IArm.moveDown? (Speed);
24  protocol
25    SAVEENERGY ≡ begin → BEHAVE;
26    BEHAVE ≡ (MAINTAIN_NB_CONTAINER + CHECKSPEED) → BEHAVE + end;
27 end OPTIMIZATION aspect saveEnergy;
```

Listing 3.19: The `saveEnergy` aspect in PRISMA AOADL

An aspect weaving in PRISMA is explicitly specified at architectural elements' specification phase, where both functional and other concerns of each architectural element are determined. Listing 3.20 shows the specification of the `Controller` component as a PRISMA architectural element. According to the specification, the `Controller` component has three concerns: a functional concern implemented by the `Controller` module (line 2) and two optimization concerns implemented by the `saveEnergy` and the `truckSafety` modules (line 3-4). The weaving is indicated with the `weavings` keyword (line 6-11). The weaving says that the `load` service of the `truckSafety` module is executed before the `load` service of the `saveEnergy` module (line 7) which in turn executed before the `load` service of the `Controller` (line 8); and the same for the `unload` service (line 9-10). In this case the `before` operator is used. In addition, PRISMA defines other similar operators for services composition such as `after`, `instead`, and `beforeIf`. Indeed, to complete the weaving of the `saveEnergy` and the `truckSafety` aspects, the same aspect modules should be imported by the `Engine` component specification and their services execution should

be ordered in the weaving part of the specification.

```
component Controller
  FUNCTIONAL aspect import Controller;
  OPTIMIZATION aspect import saveEnergy;
  OPTIMIZATION aspect import truckSafety;

  weavings
    truckSafety.load() before saveEnergy.load();
    saveEnergy.load() before Controller.load();
    truckSafety.unload() before saveEnergy.unload();
    saveEnergy.unload() before Controller.unload();
  end weavings;

  ports
    iCtrl: IController;
    iEng: IEngine;
  end ports
end component;
```

Listing 3.20: Weaving the `saveEnergy` aspect on the `Controller` component in PRISMA

Limitations

In PRISMA, aspects are considered from the design stage as regular concerns that may crosscut other concerns. However, no pointcut language is provided, and hence the designer should be aware of all the join points at the beginning listing them one by one. Moreover, PRISMA provides an advice level composition strategies for aspects using a set of keywords, using those keywords, a developer must indicate the explicit execution order of advices for each join point which is not practical solution, and hence more aspect level composition strategies are needed. The main advantage of PRISMA, is its formal model support of protocol specification of concerns which allows a validation of architectural elements properties.

3.4.2 AspectLEDA

AspectLEDA approach [Navasa 2009] defines aspects at the design stage. First, UML diagrams are described for the base system without aspects. Second, aspects are added to the system by maintaining the previously defined diagrams to deal with aspects. At this stage, designers extract information about aspects interactions with the base system. These information include, the interaction points of each aspect, the components involved in the interaction and when each aspect is applied. The extracted information are used later at the architectural design stage where the architecture of the base system with aspects is described in an AOADL provided by the framework. Aspects in AspectLEDA are regular components implementing the advices code and expose them with a regular provided interface. Aspect weaving is modeled as binding aspect components to base system components by means of *coordinators*. A coordinator monitors base system components and calls the aspect

advices services when a join point is reached. In AspectLEDA, a coordinator is defined for each join point and when several aspects shares a join point, a *Supercoordinator* is interrogated to get the order in which aspects must be executed based on predefined priority rules.

```
1 component extendedCraneSystem {
2   composition
3     craneSystem : System;
4     saveEnergy : Aspect;
5     truckSafety : Aspect;
6   priority truckSafety > saveEnergy;
7   attachments
8     craneSystem.User.load() ≪Controller, RMA, NoAvail, before≫
9         saveEnergy.maintainNbContainer();
10    craneSystem.User.load() ≪Controller, RMA, NoAvail, before≫
11        truckSafety.setLoadingState();
12    craneSystem.User.unload() ≪Controller, RMA, NoAvail, before≫
13        saveEnergy.maintainNbContainer();
14    craneSystem.Engine.moveUp(Speed) ≪Arm, RMA, NoAvail, before≫
15        saveEnergy.checkeForSpeed();
16    craneSystem.Engine.moveDown(Speed) ≪Arm, RMA, NoAvail, before≫
17        saveEnergy.checkeForSpeed();
18    craneSystem.Engine.moveDown(Speed) ≪Arm, RMA, NoAvail, before≫
19        truckSafety.checkeForSpeed();
20 }
```

Listing 3.21: The crane system extended with `saveEnergy` and `truckSafety` aspects in AspectLEDA AOADL

Listing 3.21 shows the crane system extended with `saveEnergy` and `truckSafety` aspects in AspectLEDA. The `composition` section line (2-5) defines the base system, and the set of the aspects to be added. The `priority` clause indicates that the `truckSafety` is executed before the `saveEnergy` aspect. This rule is used by the Supercoordinator when shared join points are captured (*i.e.,* `load` and `unload` in our case) by a coordinator. The interaction between a component system and an aspect is described in the `attachments` section. Each attachment defines: the component calling the service, the receiver component, the type of the event, the condition in which the aspect is executed, and when the aspect is executed with respect to the captured event. For example, The attachment description at line (8-9) says that whenever the `load` method of the `User` component is intercepted (RMA: Receive messAge) the `maintianNbContainer` method of the `saveEnergy` aspect component is always (NoAvail: Not Available condition) executed before (before keyword).

Limitations

AspectLEDA provides a primitive pointcut language (*i.e.,* call, receive) and very limited advice execution strategies (*i.e.,* before, after, conditional). The model does not provide any form support for interference detection, it is up to the designer to detect potential interferences from the sequence diagrams of case studies extended with aspect behaviors.

3.5 Lessons learned

The above sections overviewed a set of component models and their support of aspect-oriented mechanism. In this section we review those models with respect to a set of criteria we propose for a generic support of aspect orientation in component based systems. Fulfilling all the proposed criteria enables better modularity, reusability and encapsulation of both components and aspects before and after weaving.

Hierarchy: By hierarchization, different components can be encapsulated by one composite component. In our opinion, a composite should not be only an abstraction and an encapsulation of a set of components. Since external calls pass through the composite, it is natural that composites implement their own behavior and have the ability to accept or reject external calls. This allows aspects to be implemented in a unique module and integrated at the composite level to be applied to all its inner components. From the above discussed models, only PRISMA and Fractal support hierarchization.

Pointcut Language: The pointcut language should be expressive enough to define the set of interesting points that should be captured at runtime. Current models just project object-oriented pointcuts to the world of components. We understand that its is due to the fact that most component models are object oriented ones. However, specific pointcuts for components enables the definition of join points in a declarative way. For example, FAC extension of Fractal enables `CLIENT` and `SERVER` keywords to indicate the set of required and provided interfaces by a component. These are specific pointcut elements that are used rather than specifying explicitly one by one the different interfaces implemented or used by a component.

Interference Detection support: Aspect interference detection is a main feature that a component model with aspects should provide. Without this support, there is no guarantee that the added aspects works properly together when they are woven to the same system. A component model developer should be aided with this support to indicate the emergence of interferences and possibly indicate what to do to solve the conflict. In AspectLEDA, aspects are detected at the design stage and their weaving require updating the UML sequence diagrams of the base system; at this stage developers can detect possible interferences. However, UML sequence diagrams only enables the detection of structural interferences and not behavioral ones (*i.e.,* interferences at shared join points). Note that interference detection is still a challenge in AOP and none of the above models provides a support to this feature.

Aspect Composition: when interferences are detected, a mechanism for solving these interferences should be provided. One of possible solutions is aspect ordering that indicates that one aspect should be executed before another.

AspectCCM, FAC and SpringAOP enables this strategy inspiring it from AspectJ. In fact, aspect ordering specifies the same execution order of their advices at shared join points, but in some cases the order should not be the same for all the intercepted join points. To overcome this problem, AES, CAM/DAOP, JBossAOP and AspectLEDA enables advice ordering. However, coarse-grained ordering (*i.e.,* aspect ordering) can be used in conjunction with fine-grained ordering (*i.e.,* advice ordering), for example, when aspects share one or more join points and define a large set of advices that should be executed within the same order, aspect ordering is useful for simplification. In addition, aspect composition goes beyond ordering, especially when interferences are due to the weaving of aspects with no shared join points. It is the case for the `saveEnergy` and the `performance` aspects of our crane example. All the above discussed models are limited to aspect or advice ordering, but JAsCo. JAsCo supports both aspect and advice ordering, it also provides a programmatic way to develop more complex composition strategies.

Aspect Modularity: aspect modularity is the ability to define the aspect behavior in a separate module and preserves this modularity after weaving. All the above discussed models consider aspects as first class entities (*i.e.,* components, controllers, connectors, etc.), however preserving this modularity after weaving strongly depends on the weaving strategy supported by each model. The weaving shows how aspects are integrated into components. In AOP two famous strategies are provided: *static* and *dynamic weaving*. In the world of components, static weaving is not always possible because components are often available in binary form. On the other hand, dynamic weaving is technically more complex than static weaving, in addition, dynamic weaving is not possible for encrypted or digitally signed components. Moreover, enabling aspects to be woven and unwoven at runtime makes static and dynamic weaving strategies useless. In the world of components, wrapping is the strategy that can be used whatever the kind of components and enable weaving and unweaving aspects as needed. By wrapping, aspects should be modeled as first class entities and the underlying components should be monitored so that when join points are reached the aspects are called. Some of the above discussed models supports static weaving such as PRISMA, AspectLEDA, SpringAOP, and hence the modularity of aspects is not preserved after weaving by those models.

Component Instantiation: with component instantiation, several aspect instances may appear in the same system. This is an important when an aspect defines one or more state variables that are updated when join points are reached. In that case, only relevant components share the same aspect instance and state variables of the aspect are updated only for the defined components. Most of the shown models support aspect instantiation. However, the closed container-based models such as CCM and EJB, give no information about aspect instantiation either because the supported aspects are stateless ones or

the instantiation is automatically managed by their black box containers.

To sum up, Table 3.1 presents a cumulative and a direct comparison of the different component models shown in this chapter. The comparison is based upon the set of AOP criteria presented above. As shown in the table, none of the models provides a full support of all the criteria we define for a complete, generic and effective support of aspects in component models, which strongly motivates the work of this thesis.

	Hierarchy	Pointcut Language	Interf. Detection	Composition Level		Modularity		Inst.
				Aspect	Advice	Before	After	
EJB	-	-	-	-	-	+	+	?
AES	-	-	-	-	precedence	+	-	+
CCM	-	-	-	-	-	+	+	?
AspectCCM	-	-	-	precedence	-	+	+	+
Fractal	+	-	-	-	-	+	-	+
Safran	+	-	-	-	-	+	+	+
Fractal-AOP	+	-	-	-	-	+	+	+
FAC	+	CLIENT, SERVER	-	precedence	-	+	+	+
JAsCo	-	-	-	+	+	+	+	+
CAM/DAOP	-	-	-	-	precedence	+	+	+
Spring AOP	-	-	-	precedence	-	+	-	+
JBoss AOP	-	-	-	-	precedence	+	-	+
PRISMA	+	-	-	+	-	+	-	+
AspectLEDA	-	-	UML seq. diagram	-	precedence	+	-	+

Table 3.1: Current Aspectualized component models and their support level of aspects: (+) fully supported feature, (-) non supported feature, (?) unknown

Part II

Contributions

Chapter 4

Aspects as wrappers on views of component systems architectures

Contents

In this chapter, we describe our approach based on views and wrappers for aspectualizing component models. We define a declarative pointcut language VIL adopted for component models to define views on a declarative style. We introduce the basic concepts of the approach: *views* and *wrappers*, we show how they are used for the running example, and we detail how they can be implemented for two component models: Fractal and EJB.

4.1 Aspects as wrappers on views

In the CBSE development process, component assemblies are defined by software architects who decide which components are used and how they are connected.

Software architects use requirements specification to choose the appropriate configuration (assembly). In practice, different system configurations are possible, each of which meets one or more requirements. Based on this observation, we use the term *view* to refer to a component system configuration adopted for a purpose (*i.e.,* base system). In addition, we use the term *wrapper* to refer to an entity that encapsulates a component, intercepts its ingoing and/or outgoing service calls, executes extra-code, and explicitly proceed or skip original calls (*i.e.,* aspect). For better understanding of views and wrappers, let us consider the crane system and the different requirements described in Chapter 1. Here, we recall the set of those requirements, and we show how they can be fulfilled using views and wrappers:

Performance:
Enforce the crane to move fast, whatever was the running mode chosen by the user, when the arm is not carrying a container. This considerably improves the general response time of the crane.

Recovery:
Return both the engine and the arm to their stable position in the middle whenever an undesirable sequence of actions is captured. This ensures the viability of the crane system.

Truck Safety:
enforce the arm to move slow, whatever was the running mode chosen by the user, when the crane is loading a container on the truck. This ensures the safety of both the truck and the containers.

Save Energy:
enforce the arm to move slow, whatever was the running mode chosen by the user, after carrying a given number of containers. This ensures a better energy consumption of the crane.

Crane Safety:
Ignore user commands when the temperature of the engine or the arm reaches a critical value. This ensures the safety of the crane devices.

Real-Time:
Check whether loading/unloading containers is achieved in t_{speed} ($\leq t_{speed}$) time. If it is not the case, the arm must be moved up and all the subsequent requests must be refused.

Each of the above requirements adds a new functionality to the crane. In a black box based models, the above requirements cannot be added directly to the behavior of components since their source code is not available. Aspect-oriented approach tackles this problem by encapsulating the required functionalities in separate modules (*i.e.,* aspects) that can be added to the base system via aspect weaving. Note that static and dynamic weaving strategies are not always possible: static weaving is not possible if the source code of components is not available, and dynamic

weaving is not possible if the binary form of components code is encrypted or digitally signed. For this purpose, we propose to model aspect weaving as component wrapping. Thus, we model aspects as wrappers on specific views that encapsulate one or more components, intercept their ingoing and/or outgoing service calls and execute the aspect behavior when calls are intercepted. In the following, we show how views and wrappers can be used in order to force the crane system to fulfil the above requirements.

Implementing the performance aspect

The crane system can be forced to fulfill the performance requirement by adding a wrapper that surrounds the engine and the arm components. The added wrapper intercepts calls on the provided interface of the engine (*i.e.,* `iEngine` interface) and the required interface of the arm (*i.e.,* `iMagnet` interface). The wrapper stores and updates the state of the magnet whenever `setOn()` and `setOff()` services are called on the `iMagnet` interface. Thus, whenever the wrapper intercepts `moveLeft(Speed)` and `moveRight(Speed)` calls on the `iEngine` interface, it first checks the current state of the magnet. If the magnet is off the wrapper forces the engine to run in fast mode by proceeding the intercepted call with fast as a value of the parameter of the call. Adding the wrapper likewise requires the engine and the arm to be in the same composite, this is already satisfied by the provided configuration of the system and hence no reconfiguration is needed. Figure 5.1 depicts the required view for the performance wrapper. In the figure, the performance wrapper surrounds the crane composite, it is shown as a gray box surrounding the crane component.

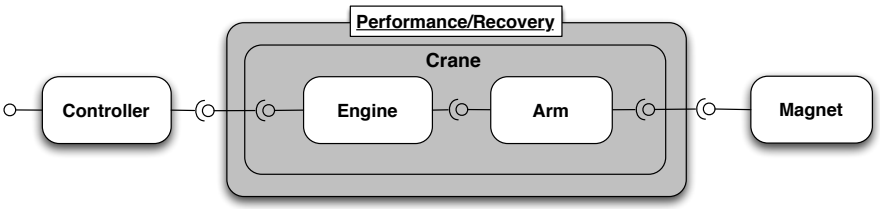

Figure 4.1: Performance/Recovery view of the crane

Implementing the recovery aspect

The recovery aspect returns both the engine and the arm to their stable position in the middle whenever an undesirable sequence of actions is captured. This requires the interception of all the actions taken by the engine and the arm to detect the undesirable sequence of actions and starts a recovery process accordingly. Thus, all the provided and required interfaces of the engine and the arm should be intercepted and hence the same view for the performance is required (see figure 5.1).

Implementing the truck-safety aspect

The truck-safety can be integrated as a wrapper around the control and the engine components. This way, the integrated wrapper can intercept calls on the provided interface of the controller (*i.e.,* `iController` interface) and the required interface of the engine (*i.e.,* `iArm` interface). The wrapper stores and updates the state on which the controller is under loading or unloading a container when it intercepts `load()` and `unload()` calls on the `iController` interface. So that, whenever the second call of `moveDown(Speed)` on the `iArm` interface is intercepted and the controller is being loading a container, the wrapper proceeds the `moveDown(Speed)` call in slow mode by setting the parameter value of the call to slow. However, the provided configuration does not match the required one (*i.e.,* the controller and the engine are not in the same composite). In this case, the system should be reconfigured to fulfil the required view as shown in Figure 5.2.

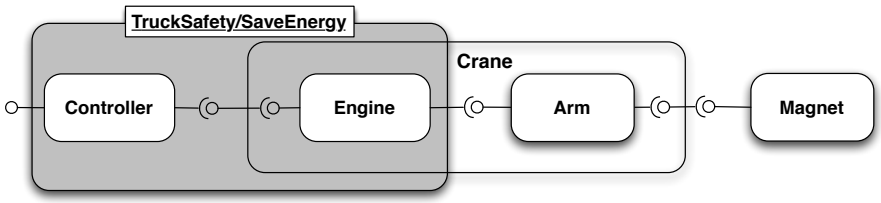

Figure 4.2: TruckSafety/SaveEnergy view of the crane

Implementing the save-energy aspect

The save-energy wrapper requires to intercept the provided interface of the controller (*i.e.,* `iController` interface) to count the number of load and unload calls. In addition, it requires the interception of the calls on the required interface of the engine (*i.e.,* `iArm` interface) to change the speed of the taken actions when the number of carried containers reaches a threshold number. The required view for this feature is the same as the one required by the truck safety (see figure 5.2).

Implementing the crane-safety aspect

The cane-safety control the temperature of the engine and the arm. The temperature of the engine and the arm are captured by their corresponding components and available through the access to their attributes and hence the engine and the arm should be in the same composite. Moreover, When the temperature of the devices reaches a provided critical value, all the requests to load or unload a container must be skipped and a message is sent to the supervisor. This requires to intercept calls on the provided interface of the controller to skip the controller action requests. As a result, the required view should encapsulate the controller, the engine, and the arm in the same composite. The required view is depicted in figure 5.3.

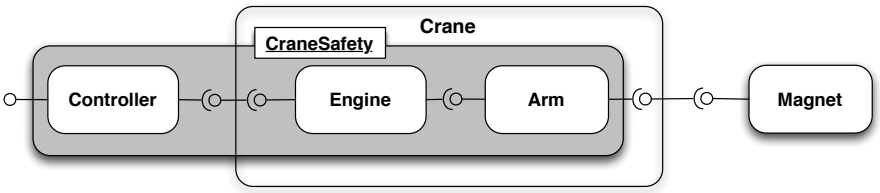

Figure 4.3: CraneSafety view of the crane

Implementing the real-time aspect

Real-time constraints can also be implemented as a wrapper on a crane system view. If the crane is in slow mode, then no real-time constraints are enforced. If the crane is in fast mode, then the time between the crane request to load/unload a container and setting the magnet off have to be achieved in t_{speed} time units. Otherwise, the arm should be moved up for its safety and all the subsequent user commands are refused and the supervisor must be warned. The required view in this case is the provided interface of the controller (to initialize the local time for the action when load or unload requests are intercepted), the required interface of the arm (to capture the interception time of setting the magnet off) and the provided interface of the arm (to call the arm to move up when the real-time constraint is violated). The corresponding view is shown in figure 5.4.

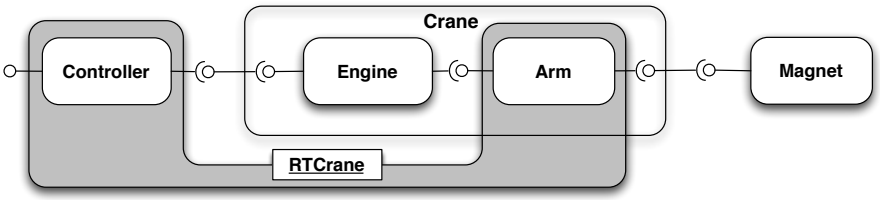

Figure 4.4: RTCrane view of the crane

As shown from the above examples, views make it simple to implement the above aspects: if the components are properly assembled (*i.e.,* the current configuration matches the required view), aspects are added as wrappers to the view otherwise a reconfiguration is needed. However, when we consider aspects requiring different views at the same time, wrappers (*i.e.,* aspects) crosscut each other as shown in Figure 5.5. It is obvious that the architecture of the component system must be reconfigured in order to enable different wrappers requiring different views. In the following, we introduce a specialized language for views definitions and show how it can be used in order to weave crosscutting wrappers.

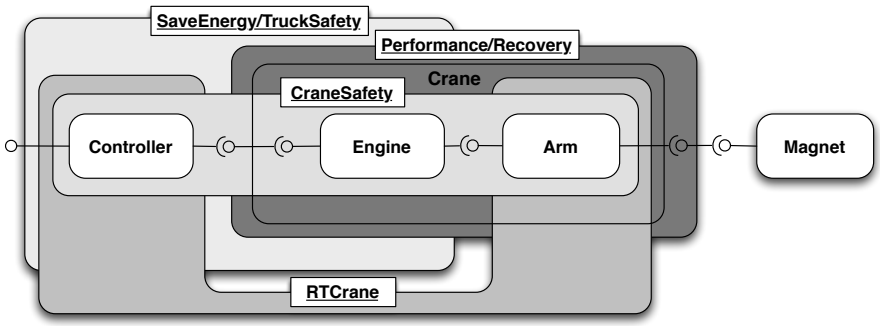

Figure 4.5: Wrappers crosscutting phenomenon

4.2 Views definition language

In this section we define a declarative language we call VIL (VIews Language) for the definition of views in component architectures. In the AOP point of view, VIL is a declarative pointcut language suitable for component models. Using VIL, minimum language constructs are used to define the points where aspects should interact with the system and this prevents the knowledge of the complete and detailed structure of the component systems. Here, we first define the join point model, then we introduce the complete syntax of the language and we describe its semantic.

4.2.1 The join point Model

The main element of each aspect-oriented language is the join point model. The join point model defines well-defined points in the execution of programs. These points are captured at runtime and aspects get involved only on those points [Kiczales 2001a]. On the other hand, components can be seen as black, gray or white boxes. Black box components are characterized with no specific knowledge of their internal behaviors, no access to their source code, and no knowledge of their architecture. In that case, the only accessible information of a component are its provided and required interfaces. Gray box components may show their internal architecture and potentially some knowledge of their behavior in terms of protocols or contracts [Szyperski 2002]. In particular, Fractal [Bruneton 2004] is a hierarchical component model where the architecture of composite components are given. Finally, white box components provides full access to their structure and behavior (*i.e.,* source code available). Thus, each of the above different categories of components requires an appropriate join point model when aspect-orientation is required. In our approach, we focus on the intermediate level (gray box component models) where components, their architectures and their behaviors specification are provided. While component architecture description enables the definition of views and the exact points where aspects are involved in the interaction with the base system, bahavior's specification enables aspect interferences detection. In our proposal, we consider calling services from required or provided services, and accessing component attributes. Inspired from

AspectJ [Kiczales 2001b], each service call at runtime is a different join point, and each component attribute access is a different join point. A join point has one of the forms `(cId, itfId, svId)` (for service call) and `(cId, atId)` (for attribute access) where `cId` ranks over component names, `itfId` and `atId` rank over interface and attribute identifiers, respectively, and `svId` ranks over service signatures. For example, the `(engine, iArm, moveUp(Speed))` denotes calling the `moveUp(Speed)` service of the `iArm` interface of the `engine` component, while `(engine, speed)` denotes the access to the attribute named `speed` of the `engine` component. The following section gives the complete syntax of VIL.

4.2.2 Syntax of VIL

In this section we describe our declarative pointcut language VIL for component models.

$$
\begin{array}{lcl}
vexp \in View & ::= & * \\
& | & cId \\
& | & \textbf{child}\ [^{+}]\ v \\
& | & \textbf{parent}\ [^{+}]\ v \\
& | & \textbf{primitive}\ v \\
& | & \textbf{instance}\ v \\
& | & [\textbf{direct}]\ \textbf{provide}\ v\ [(\textbf{T} \mid \textbf{N})\ id^{*}][sig^{*}] \\
& | & [\textbf{direct}]\ \textbf{require}\ \ v\ [(\textbf{T} \mid \textbf{N})\ id^{*}][sig^{*}] \\
& | & \textbf{bound}\ [\textbf{C} \mid \textbf{S}]\ v\ [id^{*}] \\
& | & \textbf{attributes}\ v\ [atId^{*}] \\
& | & \textbf{scflow}\ v \\
& | & v_1 \oplus v_2 \\
& | & v_1 \otimes v_2 \\
& | & v_1 \ominus v_2
\end{array}
$$

VIL defines a view in a component architecture specifying the set of components to be encapsulated together and their interfaces and services to be intercepted. In particular, "*" denotes that all the components of the architecture should be encapsulated in one component and all their interfaces and services are intercepted. Thus, the required view is the same as the original one (*i.e.,* no need to reconfigure the system) and the wrapper is to be plugged into the root component of the architecture. As described above, `cId` refers to a component identifier, this indicates that the required view is the same as the original one and all the interfaces of `cId` are intercepted, but in this case the wrapper is to be plugged into the component `cId`. **child** v and **parent** v denote that the view should encapsulate all the inner components (resp. parents) of v in the same component and all their interfaces are intercepted. "+" denotes a recursive closure of the above relations. **primitive** v denotes a view where all the primitive components of v should be in the same composite and all their interfaces are intercepted. **instance** v denotes a view where all the instances of v should be wrapped and all their interfaces are intercepted. **provide** v and **require** v denote the same view as v however, only the provided (rep. required) interfaces are intercepted. For more exhaustive specification we denote name (**N**) or type (**T**) patterns of the interfaces to be intercepted and potentially

the service signatures to be captured (*sig*). **bound** v denotes all the bound components to v. To be more precise we provide **C** and **S** to denote only the components bound to a client or to a server interfaces of v, respectively. An identifier pattern can also be used to select only the components whose names match a pattern identifier. **attributes** v denotes a view that encapsulates all the components in v and intercept accesses to all or some of their attributes. We also use **scflow** v to define a view that encapsulates all the components of v in the static control flow of the join points defined by v. Finally, a view can be composed of two other views using a union ($\oplus$), an intersection ($\otimes$) or a difference ($\ominus$) operators over views.

4.2.3 Semantics of VIL

Component architectures can alternately be represented as directed labeled graphs. Nodes in those graphs are the main architectural elements of the component architectures (*i.e., components, attributes, interfaces and services*), and the arc's labels denote the relationships among such elements (*e.g., child, provides, requires, boundTo*). These representation enables the introspection of component architectures and determine the location of elements of interest using an abstract language. FPath [David 2009b] is a domain specific language introduced to introspect Fractal architectures. FPath is a general and an extendible language and hence can be generalized for different component models by introducing new architectural elements and new relationships among elements. In the following we overview FPath and we show how our VIL expressions are evaluated using such a language.

4.2.3.1 FPath Query Language

FPath [David 2009b] is a query language developed to deal with the introspection of Fractal component architectures. FPath uses declarative path expressions to introspect Fractal elements: components, interfaces and attributes. FPath is jointly used with a scripting language named FScript to define complex runtime reconfiguration of Fractal architectures rather than using Fractal API which is a tedious and error prone task. FPath expressions are of the following form, where *nodeId* is a node identifier denoting the starting point from which the navigation starts and *arcId* denotes a transition label identifier denoting the axis to follow:

$$\begin{array}{lll} \mathit{fExp} & ::= & \$\mathit{nodeId}\ \text{“/”}\ \mathit{step}\ (\text{“/”}\ \mathit{step})^* \\ \mathit{step} & ::= & \mathit{arcId}\ \text{“::”}\ (\mathit{nodeId} \mid \star)\ [\mathit{predicate(.)}] \end{array}$$

For example, the FPath expression "*$crane/child::*/requires::**" returns the set of required interfaces (the arc label $requires :: *$) of all the inner components ($child :: *$) of the crane composite component (*$crane*). However, the current version of FPath cannot directly be used in our proposal for several reasons: (1) FPath expressions return primitive elements: a set of components or interfaces or attributes while in our join point model we need to define a set of tuples defining the interface services of components to be intercepted. (2) FPath does not consider

low level architectural elements such as services. (3) some of the current relationships are related to Fractal (*e.g.,* internal interfaces) where we need an independent language. For the above mentioned reasons, we adapt FPath to fulfil our requirements. Figure 5.6 depicts the architectural elements and the relationships required by our model. In the figure, the architectural elements considered are depicted with named circles: *component*, *interface*, *service* and *attribute*. The relationships among the elements are represented by labled arrows: *parent*, *child*, *provides*, *requires*, *has*, *boundTo*, *defines*, *set* and *get*. The elements and the relationships annotated with $\star$ are the ones newly added to FPath.

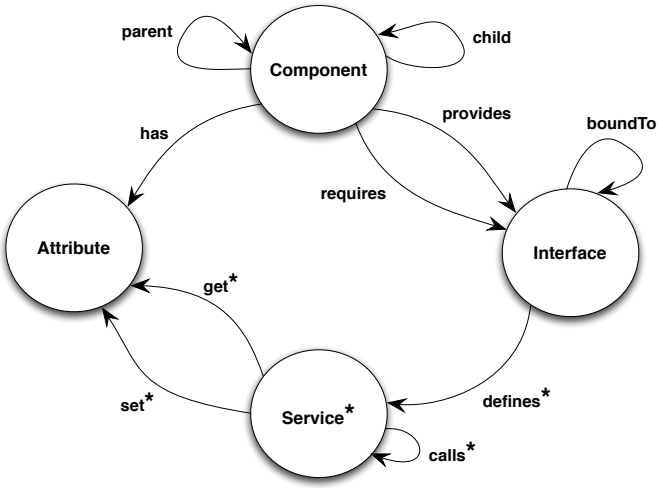

Figure 4.6: Directed labeled graph adopted for component architectures

4.2.3.2 VIL semantics in FPath

In this section we give an excerpt of the semantics of VIL, the rest can be straightforwardly deduced. VIL uses FPath to access the different architectural elements on a component architecture, and defines the different join points required to be intercepted. In addition, a views transformation function is defined for VIL to generate the required views (*i.e.,* a view where all the components in the jon points are wrapped). First we introduce a function *eval* that defines the set of join points from VIL expressions for a component architecture a. Let us consider the primitive expression cId, the *eval* function calls FPath ($\mathcal{F}_{path}$ function) to get all the interfaces of cId (*e.g., itfs*) and all the services defined for each interface (*e.g.,* svs_{itf}), finally it defines the set of join points as a set tuples $\{(cId, itf, sv) : itf \in itfs \wedge sv \in svs_{itf}\}$. Formally:

$$eval\ (cId)\ a \quad = \quad \textbf{let}\ e_1 = \$\texttt{cId/provides} :: *,\ e_2 = \$\texttt{cId/requires} :: *,\ itfs = \mathcal{F}_{path}[\![e_1]\!]\ a \cup \mathcal{F}_{path}[\![e_2]\!]\ a\ \textbf{in} \bigcup_{\forall itf \in itfs} \{(cId, itf, s) :\ s \in \mathcal{F}_{path}[\![(\texttt{e}_1|\texttt{e}_2)[\texttt{name}(.) = \texttt{itf}]/\texttt{service} :: *]\!]\ a\}$$

Another example is the expression (**child** v) which is evaluated by evaluating v first and for each component in v we get all its direct inner components capturing all their services of all their provided and required interfaces. The $s_{\downarrow_t}$ in the formula denotes a projection operation that extracts only the t elements from the tuples defined by s.

$$eval\ (\textbf{child}\ v)\ a \quad = \quad \textbf{let}\ cs = (eval\ v\ a)_{\downarrow_{component}}\ \textbf{in} \bigcup_{\forall cId' \in (\forall cId \in cs : \mathcal{F}_{path}[\![\$\texttt{cId/child}::*]\!])} eval\ cId'\ a$$

The expression (**attributes** v) is evaluated by applying the *eval* function to v capturing all the attributes defined on all the components of v.

$$eval\ (\textbf{attributes}\ v)\ a \quad = \quad \textbf{let}\ cs = (eval\ v\ a)_{\downarrow_{component}}\ \textbf{in} \bigcup_{\forall cId \in\ cs} \{(cId, at) :\ at \in \mathcal{F}_{path}[\![\$\texttt{cId/has} :: *]\!]\ a\}$$

A more interesting example is the (**scflow** v) which is evaluated by evaluating v capturing the set of join points on the recursive closure of `calls` relation of each service on v.

$$eval\ (\textbf{scflow}\ v)\ a \quad = \quad \textbf{let}\ jps = eval\ v\ a\ \textbf{in} \bigcup_{\forall sv \in\ jps_{\downarrow_{service}}} \{(cId, itf, s) :\ s \in \mathcal{F}_{path}[\![\$\texttt{sv/calls}^+ :: *]\!]\ a\}$$

A last example is views composition operations that forms one view from two predefined views. Here is the definition of the union operation of views, that constructs a new view that encapsulates all the components and intercepts all the join points encapsulated and intercepted by its underlying views.

$$eval\ (v_1\ \oplus\ v_2)\ a \quad = \quad eval\ v_1\ a\ \cup\ eval\ v_2\ a$$

In addition to the above *eval* function, we provide views transformation function σ_{mId}. This function is a model-independent function. It defines how wrappers and

views are represented according to the basic tenets of a component model mId. In the following, we show how σ_{mId} is defined to model views and wrappers for two different categories of component models: hierarchical component models with component sharing such as Fractal and flat component models such as EJB.

4.3 Implementation of VIL in Fractal component model

Fractal component model [Bruneton 2004] defines its own architecture description language (ADL) to describe component assemblies. It supports hierarchies, component controllers, introspection and component sharing. Fractal controllers intercept calls to a component provided and/or required interfaces and enable component behavior adaptation and the definition of extra-functional features to components by executing an additional code when calls are intercepted. We get benefit of Fractal controllers to define aspects (*i.e.,* wrappers). However, in Fractal Julia implementation, when a component has several controllers, there is no general way to compose them. They can only be executed independently or sequentially by configuring interceptors or be composed in a programmatic way by explicitly calling one controller from another. This makes the implementation of controller-based adaptations a complex task with sometimes unexpected behavior. To tackle this limitation, we introduce composable controllers into Fractal Julia implementation. Composable controllers, as their name indicate, get benefits of regular Fractal controllers (control the behavior of components by introducing extra-functional properties) and enable controller composition. In the following, we introduce the concept of composable controllers, we describe how views are defined using VIL, and we show how aspects (*i.e.,* controllers) can be composed to solve potential interferences.

4.3.1 Composable controllers

We define a composable controller as a pair `(Dispatcher,ICController)` where `Dispatcher` is a regular Fractal controller plugged into the composite defining the view and `ICController` is an object implementing the `ICController` interface (see Listing 5.1).

```
1 enum Action {Proceed, Skip}
2 interface ICController {
3   Action match(MessageContext c);
4 }
```

Listing 4.1: `ICController` interface

The `Dispatcher` controller intercepts calls to the inner components of the view, reifies the intercepted calls into `MessageContext` objects and calls the `match()` method of the `ICController`. The `Dispatcher` waits for the action taken by the `ICController` (*i.e.,* `Proceed` or `Skip`) and behaves accordingly. When it receives a `Proceed` action, it calls the original method and the call reaches its target, when it

receives a `Skip` the `Dispatcher` does nothing and the call is ignored. For composing controllers, composition operators are defined as `ICController`(s), this enables the composition of controllers and composition operators in a composite pattern way. Each `match(MessageContext)` method of the composition operator implements the semantics associated to each operator and returns `Proceed` or `Skip` accordingly to the `Dispatcher` controller. Take for example the `crane` view of the crane system (see Figure 5.7).

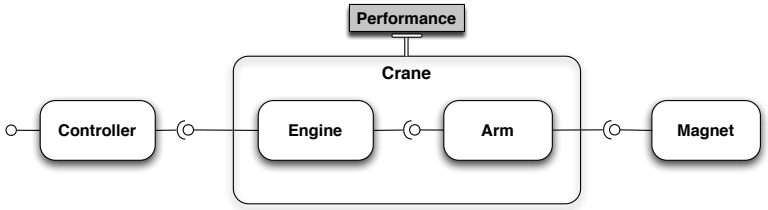

Figure 4.7: A composable controller on the `Performance` view of the crane where the `ICController` is shown as a gray box with the name of the aspect and the `Dispatcher` is depicted with (⊤) at the top of the view

The integration of the `Performance` aspect to the crane requires adding a composbale controller `(Dispatcher,Performance)` to the original view. The `Dispatcher`, in this case, intercepts the incoming calls to the `Engine` and the outgoing calls from the `Arm`. For each intercepted call, the `Dispatcher` reifies it and calls the `match()` method of the `Performance` object (see Listing 5.2). The `match()` method of the `Performance` checks whether the intercepted call is a `setOn()` (resp. `setOf()`) method call (line 6-7). If it is the case, it stores the state of the magnet in the attribute `isMagnetOn`. If the intercepted call is a `moveLeft(Speed)` or a `moveRight(Speed)` method call (line 8), it first checks the state of the magnet, if it is off, it changes the parameter value of the call to `fast` by calling `c.setSpeedArgument("fast")` method (code not shown here). Finally, the `match()` method returns `Proceed` (line 11) thus the `Dispatcher` calls the original method which reaches its target.

```
1 class Performance implements ICController {
2
3   private boolean isMagnetOn = false;
4
5   Action match(MessageContext c) {
6     if (c.getSignature().equals("setOn()")) isMagnetOn = true;
7     else if (c.getSignature().equals("setOff()")) isMagnetOn = false;
8       else if (c.getSignature().startsWith("move")) {
9           if (isMagnetOn()) c.setSpeedArgument("fast");
10      }
11    return Action.Proceed;
12  }
13 }
```

Listing 4.2: The `Performance` implementation

Now, we want to apply the `TruckSafety` and the `SaveEnergy` aspects, both require another view where the `Controller` and the `Engine` to be in the same component. However, since both aspects intercept the same join points, a composition

strategy should be used. In our approach we adopt the sequential composition as a default strategy for aspects sharing join points. The sequential composition is abstracted with a generic and a reusable operator named `Seq`. The `Seq` operator forwards calls to shared join points to both aspects in a sequential order and proceeds a call only when at least one of its underlying aspects returns `Proceed`, otherwise it skips the call (see Chapter 6 for the full semantics of operators). Figure 5.8 depicts the Fractal architecture of the crane example after applying both the `TruckSafety` and the `SaveEnergy` aspects (*i.e.,* `ICController`(s)). In the figure, the new view crosscuts the original view, thanks to component sharing, this configuration is possible (*i.e.,* the `Engine` component is a shared component between the two views). Shared components are depicted with dashed lines in the figure.

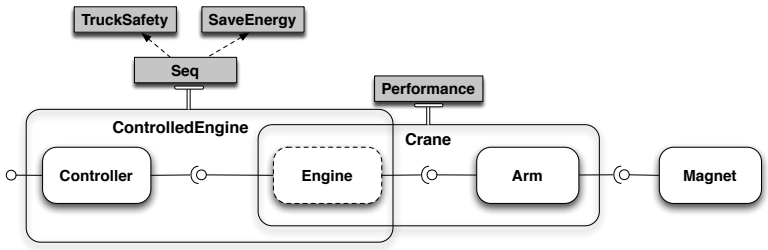

Figure 4.8: `Seq(TruckSafety,SaveEnergy)` plugged into `ControlledEngine` view

Let us generalize and detail our implementation of views for Fractal. For the definition of views in Fractal Julia, we distinguish two cases: (1) the components to be wrapped belong to the same composite, and (2) the components to be wrapped do not belong to the same composite. In the following we detail how views are created for each case.

4.3.2 The components of interest belong to the same composite

In this case, the required view is the same as the original configuration. In practice, this requires to update the ADL description of the architecture by (1) declaring a composable controller modelling the required aspect as a part of the membrane of the composite encapsulating the components of interest and (2) by declaring an interceptor controller as a part of the membrane of each component of interest. In addition, a Julia configuration file is generated to include the definition of the added controllers. Listing 5.3 shows the general required modification of the ADL description where the component named *c* models the required view by a VIL expression. In that case, a composable controller is defined for that component (`CControllerName` line 12), and an interceptor controller is defined for each one of its inner components (`Interceptor` line 10). Each interceptor controller intercepts incoming and/or outgoing calls to/from its underlying component and forwards them to the `Dispatcher` of the composable controller of its component parent.

```
1 <component name="c">
2     // interface declarations
```

```
3     // the content declaration if "c" is a primitive
4     // else for each inner component ci
5     <component name= "ci">
6       // interface declarations of the inner component
7       // the content declaration if "ci" is a primitive
8       // or the inner components declaration if "ci" is a composite
9       // binding declarations if "ci" is a composite
10      <controller desc = "Interceptor"/>
11    </component>
12  <controller desc = "CControllerName"/>
13 </component>
```

Listing 4.3: The required ADL modification if the view corresponds to the original configuration: the underlined code is the one that is added to model the wrapper

In addition to the above ADL description modification, a Julia configuration file should be defined to include the full description of each controller: its implementation class, the required interceptors and a controller composition expression if any. Listing 5.4 shows the structure of the Julia configuration file. The file starts with an indication of the composable controller identifier, the same used in the ADL description (*i.e.,* `CControllerName` line 1). The interface of the controller is defined (`ICController` line 4). The implementation class of the controller is defined by the `Dispatcher` class name followed by *ccExp* expression that is passed as a parameter to the `Dispatcher` class. The *ccExp* expression is of the form:

$$\begin{array}{lcl} \langle ccExp \rangle & ::= & ICControllerName \mid \langle opId \rangle\ (\langle ccExp \rangle, \langle ccExp \rangle) \\ \langle opId \rangle & ::= & Seq \mid Fst \mid Alt \mid ... \end{array}$$

In addition, the `InterceptorKind` is one of three kinds: `InInterceptor` that intercepts only the incoming calls of a component, `OutInterceptor` that intercepts only the outgoing service calls of a component, and `InOutInterceptor` that intercepts all the incoming and the outgoing calls of a component.

```
1 (CControllerName
2   (
3     'interface-class-generator
4     (
5       (CControllerName-controller vil.common.ICController)
6     )
7     (
8       (vil.controllers.Dispatcher <ccExp>)
9     )
10    (
11      (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
12        org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
13      )
14    )
15    org.objectweb.fractal.julia.asm.MergeClassGenerator
16    'optimizationLevel
17  )
18 )
19
20 (Interceptor
21   (
22     'interface-class-generator
23     (
```

```
24        (Interceptor-controller vil.common.IInterceptor)
25      )
26      (
27        (vil.controllers.Interceptor)
28      )
29      (
30        (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
31          org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
32          (vil.interceptors.InterceptorKind 'interfaceName)
33        )
34      )
35      org.objectweb.fractal.julia.asm.MergeClassGenerator
36      'optimizationLevel
37    )
38 )
```

Listing 4.4: The generated Julia configuration file for each view definition

For better understanding of our modelization, take for example the crane example with its crane view. Listing 5.5 shows the updated ADL description of the crane example after adding the `Performance` aspect to the crane component. The only modification is the addition of interceptor controllers `Interceptor` to the inner components of the `Crane` (line 15,21) and a composable controller named `Performance` to the `Crane` composite (line 26).

```
1 <component name=root>
2   <interface name="iController" signature="Crane.IController" role="server">
3   <component name="controller">
4     <interface name="iController" signature="Crane.IController" role="server"
          >
5     <interface name="iEngine" signature="Crane.IEngine" role="client">
6     <content class="Crane.ControllerImpl">
7   </component>
8   <component name="crane">
9     <interface name="iEngine" signature="Crane.IEngine" role="server">
10    <interface name="iMagnet" signature="Crane.IMagnet" role="client">
11    <component name=engine>
12      <interface name="iEngine" signature="Crane.IEngine" role="server">
13      <interface name="iArm" signature="Crane.IArm" role="client">
14      <content class="Crane.EngineImpl">
15      <controller desc = Interceptor/>
16    </component>
17    <component name="arm">
18      <interface name="iArm" signature="Crane.IArm" role="server">
19      <interface name="iMagnet" signature="Crane.IMagnet" role="client">
20      <content class="Crane.ArmImpl">
21      <controller desc = Interceptor/>
22    </component>
23    <binding client="this.iEngine" server="engine.iEngine"/>
24    <binding client="engine.iArm" server="arm.iArm"/>
25    <binding client="arm.iMagnet" server="this.iMagnet"/>
26    <controller desc = Performance/>
27  </component>
28  <component name="magnet">
29    <interface name="iMagnet" signature="Crane.IMagnet" role="server">
30    <content class="Crane.MagnetImpl">
31  </component>
32  <binding client="this.iController" server="controller.iController"/>
33  <binding client="controller.iEngine" server="crane.iEngine"/>
34  <binding client="crane.iMagnet" server="magnet.iMagnet"/>
```

```
35 </component>
```

Listing 4.5: The Fractal ADL description of the crane example after adding the `Performance` aspect

Listing 5.6 describes the Julia configuration file that defines the `Performance` controller. In the listing, the interface implemented by the controller (*i.e.,* `ICController`) is indicated at line 5. The `Dispatcher` controller is indicated at line 8 where the `Performance` is passed as a parameter to the `Dispatcher`. Thus, an object of the class `Performance` is created for the `Dispatcher` controller. The rest of the code in the listing indicates how the code of the controller is merged with the component implementation code.

```
1 (Performance
2   (
3     'interface-class-generator
4     (
5       (performance-controller  vil.common.ICController)
6     )
7     (
8       (vil.controllers.Dispatcher  Performance)
9     )
10    (
11      (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
12        org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
13      )
14    )
15    org.objectweb.fractal.julia.asm.MergeClassGenerator
16    'optimizationLevel
17  )
18 )
```

Listing 4.6: The Julia configuration file for the `Performance` aspect

4.3.3 The components of interest are scattered in the architecture

Here we need to reconfigure the system to accept the required view. Thanks to component sharing feature in Fractal, this kind of reconfiguration is possible. In Fractal we adopt the following reconfiguration strategy: a new composite is created as a child of the closest common parent of the required components. This composite shares all the required components with their original parents. Similar to the above case, the composable controller is plugged into the new composite and an interceptor controller is added to the membrane of each inner component. Thus, the view is added and the original configuration is preserved. The choice for the position of the new composite declaration is made to synchronize the life-cycle of the view with the life-cycle of its inner components. Thus, when all the inner components of the view are destroyed, the view is automatically destroyed. In addition, when two views crosscut each other and an interference appears between their composable controller, both views are composed into one view using the union operator ($\oplus$). This enables

the composition of the composable controllers of different views to solve potential interferences among them.

Thus, the definition of a view in this case is divided into two steps. The first step consists in finding the closest common parent of the components to be wrapped. This can be done using FPath language: consider c_{11} and c_{21} two different components belonging to c_1 and c_2, respectively. The following FPath expression provides a set of all their common parents including the root component:

$$e = c_{11}/ancestor :: *[in(c_{21}/ancestor :: *)]$$

The "$c_{11}/ancestor :: *$" sub-expression returns the set of all the ancestors of c_{11} including the root component. With the predicate "*in*" presented between square brackets, only the ancestors of c_{11} that belong to the set of ancestors of c_{21} will be returned. The closest parent c belongs to that set and has the following particularity: $descendant(c) \cap \mathcal{F}_{path}(e) = \phi$ which states that none of the descendants of the closest parent is a common parent of the required components. The second step consists in adding a new composite as an inner component of the common parent of c_{11} and c_{21} found by the previous step. This is shown in Listing 5.7. The new composite *view* (line 24-28) declares c_{11} and c_{21} as its inner components sharing them with their original parents c_1 and c_2, respectively (line 25-26). This way, the original architecture is preserved after the new view is defined. Finally, an interceptor controller is associated to each shared component (line 9,19), and a composable controller is added to the new created composite (line 27). The added interceptor controllers intercept calls on their components and route them to the nesting composite.

```
1  <component name="c1">
2      // interface declarations
3      <component name= "c11">
4        // interface declarations of the inner component
5        // the content declaration if "c11" is a primitive
6        // or the inner components declaration if "c11" is a composite
7        // binding declarations if "c11" is a composite
8        <controller desc = "Interceptor"/>
9      </component>
10     // other inner components of c1
11 </component>
12 <component name="c2">
13     // interface declarations
14     <component name= "c21">
15       // interface declarations of the inner component
16       // the content declaration if "c21" is a primitive
17       // or the inner components declaration if "c21" is a composite
18       // binding declarations if "c21" is a composite
19       <controller desc = "Interceptor"/>
20     </component>
21     // other inner components of c2
22 </component>
23 // other component declarations of this level
24 <component name = "view">
25     <component name = "c11" definition = "/c1/c11"/>
26     <component name = "c21" definition = "/c2/c21"/>
```

```
27     <controller desc = "CControllerName"/>
28 </component>
```

Listing 4.7: The required ADL modification if the components of interest are scattered in the architecture

Listing 5.8 shows the ADL description of the reconfigured crane example to include the composition of the `truckSafety` and the `saveEnergy` aspects. The added lines are underlined in the script. The Julia configuration file generated for the view includes the definition given in Listing 5.4 replacing $\langle ccExp \rangle$ with `Seq(truckSafety,saveEnergy)`.

```
1 <component name=root>
2   <interface name="iController" signature="Crane.IController" role="server">
3   <component name="controller">
4     <interface name="iController" signature="Crane.IController" role="server"
          >
5     <interface name="iEngine" signature="Crane.IEngine" role="client">
6     <content class="Crane.ControllerImpl">
7     <controller desc = Interceptor/>
8   </component>
9   <component name="crane">
10    <interface name="iEngine" signature="Crane.IEngine" role="server">
11    <interface name="iMagnet" signature="Crane.IMagnet" role="client">
12    <component name=engine>
13      <interface name="iEngine" signature="Crane.IEngine" role="server">
14      <interface name="iArm" signature="Crane.IArm" role="client">
15      <content class="Crane.EngineImpl">
16      <controller desc = Interceptor/>
17    </component>
18    <component name="arm">
19      <interface name="iArm" signature="Crane.IArm" role="server">
20      <interface name="iMagnet" signature="Crane.IMagnet" role="client">
21      <content class="Crane.ArmImpl">
22    </component>
23    <binding client="this.iEngine" server="engine.iEngine"/>
24    <binding client="engine.iArm" server="arm.iArm"/>
25    <binding client="arm.iMagnet" server="this.iMagnet"/>
26  </component>
27  <component name="magnet">
28    <interface name="iMagnet" signature="Crane.IMagnet" role="server">
29    <content class="Crane.MagnetImpl">
30  </component>
31  <component name = "ControlledEngine">
32    <component name = "controller" definition = "/controller"/>
33    <component name = "engine" definition = "/crane/engine"/>
34    <controller desc = "Seq - truckSafety - saveEnergy"/>
35  </component>
36  <binding client="this.iController" server="controller.iController"/>
37  <binding client="controller.iEngine" server="crane.iEngine"/>
38  <binding client="crane.iMagnet" server="magnet.iMagnet"/>
39 </component>
```

Listing 4.8: The Fractal ADL description of the crane example after adding the `Seq(TruckSafety,SaveEnergy)` composable controller

4.3.4 Fractal Weaver

The above component architecture transformation becomes a tedious and error prone task when the architecture grows. Our approach makes it possible to automatize this task. For that reason we developed a Fractal weaver as a top level extension of Fractal Julia implementation as shown in Figure 5.9. In the figure, the Fractal weaver consists of three modules: *VIL analyzer*, *ADL transformer*, and *Julia config generator*. In the following we describe the role of each module.

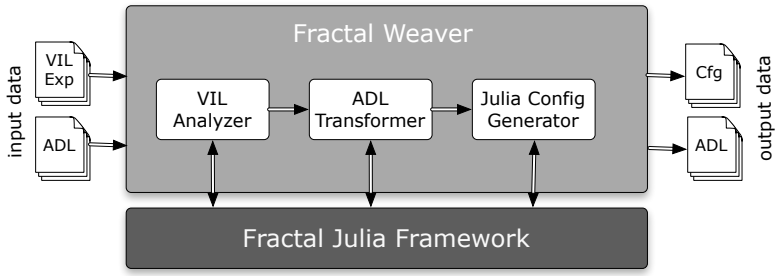

Figure 4.9: Fractal Weaver Architecture

4.3.4.1 VIL Analyzer

This module analyzes the input VIL expressions, and uses Julia introspection mechanism to introspect the given ADL description and returns the concrete join points to be intercepted. When an expression is not well-defined, an exception is thrown indicating an error in the input expression, otherwise a set of concrete join points is returned (see Section 5.2.1). Figure 5.10 describes the UML class diagram associated to VIL expressions.

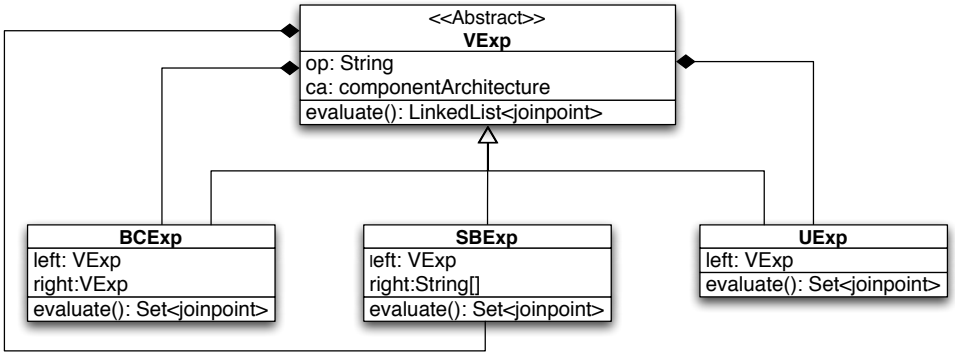

Figure 4.10: VIL Expressions structure

A VIL expression is of three forms: (1) Unary form (`UExp`) where the expression has an operator and merely one operand. This includes **child**, **parent** and **instance** expressions. (2) Selective binary form (`SBExp`) where the right hand side operand is a set of patterns or selectors that match a subset of the find join points. This includes

provide, **require** and **direct** expressions. (3) Binary composition of expressions (`BCExp`) where views are composed using $\oplus$, $\ominus$ and $\otimes$ operators. The analyzer role is to interpret VIL expressions and returns a set of join points (`Set<joinpoint>`). The `evaluate()` function of each class implements the required behavior of the analyzer for the form in question.

4.3.4.2 ADL Transformer

This module uses the returned join points from the above module and uses them to determine and define the required views. It uses the process described in Section 5.3.2 and Section 5.3.3, respectively. This includes the addition of new composites, the declaration of composable controllers and interceptor controllers for components, and the composition views when needed (see Section 5.3.3).

```
1 interface IADLTransformer {
2   Dag Adl2Dag(File src);
3   File Dag2Adl(Dag dag);
4   run(ComponentArchitecture ca, String asId, VExp exp);
5 }
```

Listing 4.9: ADL Transformer Interface

The ADL transformer uses an intermediate DAG structure to transform the ADL description into a component architecture with the required views and controllers. Listing 5.9 describes the interface implemented by the ADL transformer. It defines three methods: `Adl2Dag()` and `Dag2Adl()` to transform an ADL description into a DAG structure and vice versa, and a `run()` method that calls the `Adl2Dag()` method to get the DAG structure of the original architecture, then it calls the VIL analyzer to interpret a VIL expression, update the DAG structure to consider the required view and calls the `Dag2Adl()` method to get the final ADL description with the required views and composable controller(s).

4.3.4.3 Julia Config Generator

This latter module generates the required Julia configuration files providing information about the implementation of composable controllers and interceptors. These files are required for the correct execution of the result system in Julia. It automatizes the step described in Section 5.3.3.

4.4 Implementation of VIL in EJB component model

For flat component models, such as EJB, a view cannot be modeled as a composite since component hierarchy is not supported by such models. In addition, neither controllers nor component sharing are supported. One solution is to model a wrapper as a pair of regular components or beans `(Dispacher,Aspect)`. The `Dispatcher` bean is bound to all the beans of interest on the interfaces to be intercepted. Since the `Dispatcher` in this case has no a general structure (*i.e.,* no predefined number and type of interfaces), it can be automatically generated from the join points

definition (*i.e.,* defines all the interfaces in the set of defined join points). Like the one in Fractal, the dispatcher intercepts all the calls, defined by VIL expressions, instead of their original targets, reifies and forwards the calls to the aspect bean. The `Aspect` bean executes extra-code and decides whether to proceed or skip the call by returning `proceed` or `skip` to the dispatcher. This latter proceeds calls by calling the original service and skips them by ignoring them. Implementing this solution, the component system is reconfigured by adding the components modelling the dispatcher and the aspect bean to the container and update the bindings to fulfil the required view as depicted in Figure 5.11. In the figure, the dispatcher and

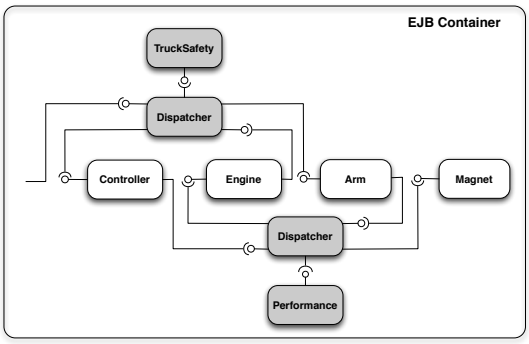

Figure 4.11: The implementation of views and wrappers in flat component models

the aspect beans are depicted with gray boxes. Two crosscutting views (`Crane` and the `ControlledEngine`) are presented. The `Dispatcher` bean in this case defines two required interfaces (*i.e.,* `IController` and `IArm`) and two provided interfaces (*i.e.,* `IController` and `IArm`). An `Aspect` bean always provides only one interface similar to that defined for Fractal (*i.e.,* `ICController` interface). This enables the composition of aspect beans using composition operator. A composition operator can also be modeled as components or beans that intercept reified join points and forward them to beans modelling the different aspects following their appropriate strategy, then return the result to the dispatcher. Figure 5.12 shows how the `TruckSafety` and the `SaveEnergy` wrappers are composed using the `Seq` operator.

4.5 Conclusion

In this chapter, we described our approach based on views and wrappers for aspectualizing component models. Views can be defined using VIL language in a declarative style. VIL is a pointcut language adopted for component models. Crosscutting views can be modeled in different ways following the basic tenets of each component model. Our approach is designed to be model and language-independent, we have shown how it can be used for hierarchical component models with component sharing such as Fractal and flat component models such as EJB. Currently, the approach is implemented for Fractal component model and the Fractal weaver presented in Section 5.3.4 is available upon request.

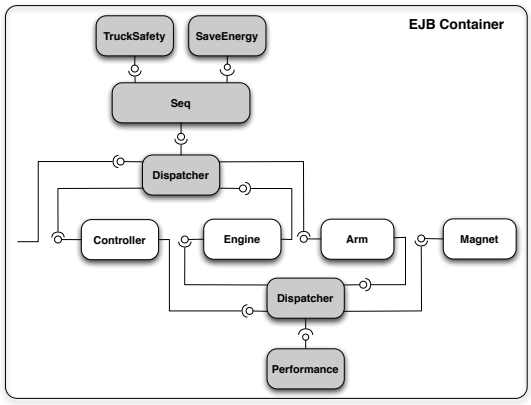

Figure 4.12: Wrappers composition in flat component models

Chapter 5

Aspects as wrappers on views of component systems architectures

Contents

In this chapter, we describe our approach based on views and wrappers for aspectualizing component models. We define a declarative pointcut language VIL adopted for component models to define views on a declarative style. We introduce the basic concepts of the approach: *views* and *wrappers*, we show how they are used for the running example, and we detail how they can be implemented for two component models: Fractal and EJB.

5.1 Aspects as wrappers on views

In the CBSE development process, component assemblies are defined by software architects who decide which components are used and how they are connected.

Software architects use requirements specification to choose the appropriate configuration (assembly). In practice, different system configurations are possible, each of which meets one or more requirements. Based on this observation, we use the term *view* to refer to a component system configuration adopted for a purpose (*i.e.,* base system). In addition, we use the term *wrapper* to refer to an entity that encapsulates a component, intercepts its ingoing and/or outgoing service calls, executes extra-code, and explicitly proceed or skip original calls (*i.e.,* aspect). For better understanding of views and wrappers, let us consider the crane system and the different requirements described in Chapter 1. Here, we recall the set of those requirements, and we show how they can be fulfilled using views and wrappers:

Performance:
Enforce the crane to move fast, whatever was the running mode chosen by the user, when the arm is not carrying a container. This considerably improves the general response time of the crane.

Recovery:
Return both the engine and the arm to their stable position in the middle whenever an undesirable sequence of actions is captured. This ensures the viability of the crane system.

Truck Safety:
enforce the arm to move slow, whatever was the running mode chosen by the user, when the crane is loading a container on the truck. This ensures the safety of both the truck and the containers.

Save Energy:
enforce the arm to move slow, whatever was the running mode chosen by the user, after carrying a given number of containers. This ensures a better energy consumption of the crane.

Crane Safety:
Ignore user commands when the temperature of the engine or the arm reaches a critical value. This ensures the safety of the crane devices.

Real-Time:
Check whether loading/unloading containers is achieved in t_{speed} *(*$\leq t_{speed}$ *) time. If it is not the case, the arm must be moved up and all the subsequent requests must be refused.*

Each of the above requirements adds a new functionality to the crane. In a black box based models, the above requirements cannot be added directly to the behavior of components since their source code is not available. Aspect-oriented approach tackles this problem by encapsulating the required functionalities in separate modules (*i.e.,* aspects) that can be added to the base system via aspect weaving. Note that static and dynamic weaving strategies are not always possible: static weaving is not possible if the source code of components is not available, and dynamic

weaving is not possible if the binary form of components code is encrypted or digitally signed. For this purpose, we propose to model aspect weaving as component wrapping. Thus, we model aspects as wrappers on specific views that encapsulate one or more components, intercept their ingoing and/or outgoing service calls and execute the aspect behavior when calls are intercepted. In the following, we show how views and wrappers can be used in order to force the crane system to fulfil the above requirements.

Implementing the performance aspect

The crane system can be forced to fulfill the performance requirement by adding a wrapper that surrounds the engine and the arm components. The added wrapper intercepts calls on the provided interface of the engine (*i.e.,* `iEngine` interface) and the required interface of the arm (*i.e.,* `iMagnet` interface). The wrapper stores and updates the state of the magnet whenever `setOn()` and `setOff()` services are called on the `iMagnet` interface. Thus, whenever the wrapper intercepts `moveLeft(Speed)` and `moveRight(Speed)` calls on the `iEngine` interface, it first checks the current state of the magnet. If the magnet is off the wrapper forces the engine to run in fast mode by proceeding the intercepted call with fast as a value of the parameter of the call. Adding the wrapper likewise requires the engine and the arm to be in the same composite, this is already satisfied by the provided configuration of the system and hence no reconfiguration is needed. Figure 5.1 depicts the required view for the performance wrapper. In the figure, the performance wrapper surrounds the crane composite, it is shown as a gray box surrounding the crane component.

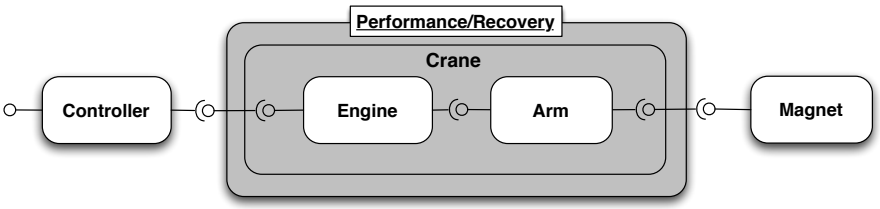

Figure 5.1: Performance/Recovery view of the crane

Implementing the recovery aspect

The recovery aspect returns both the engine and the arm to their stable position in the middle whenever an undesirable sequence of actions is captured. This requires the interception of all the actions taken by the engine and the arm to detect the undesirable sequence of actions and starts a recovery process accordingly. Thus, all the provided and required interfaces of the engine and the arm should be intercepted and hence the same view for the performance is required (see figure 5.1).

Implementing the truck-safety aspect

The truck-safety can be integrated as a wrapper around the control and the engine components. This way, the integrated wrapper can intercept calls on the provided interface of the controller (*i.e.,* `iController` interface) and the required interface of the engine (*i.e.,* `iArm` interface). The wrapper stores and updates the state on which the controller is under loading or unloading a container when it intercepts `load()` and `unload()` calls on the `iController` interface. So that, whenever the second call of `moveDown(Speed)` on the `iArm` interface is intercepted and the controller is being loading a container, the wrapper proceeds the `moveDown(Speed)` call in slow mode by setting the parameter value of the call to slow. However, the provided configuration does not match the required one (*i.e.,* the controller and the engine are not in the same composite). In this case, the system should be reconfigured to fulfil the required view as shown in Figure 5.2.

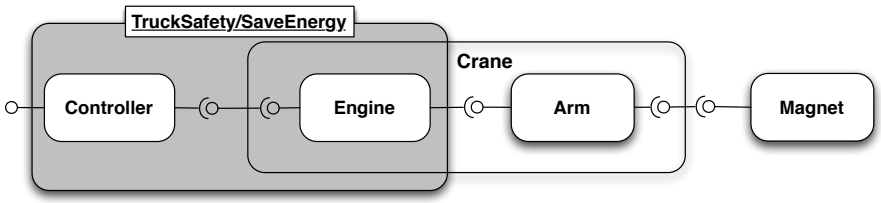

Figure 5.2: TruckSafety/SaveEnergy view of the crane

Implementing the save-energy aspect

The save-energy wrapper requires to intercept the provided interface of the controller (*i.e.,* `iController` interface) to count the number of load and unload calls. In addition, it requires the interception of the calls on the required interface of the engine (*i.e.,* `iArm` interface) to change the speed of the taken actions when the number of carried containers reaches a threshold number. The required view for this feature is the same as the one required by the truck safety (see figure 5.2).

Implementing the crane-safety aspect

The cane-safety control the temperature of the engine and the arm. The temperature of the engine and the arm are captured by their corresponding components and available through the access to their attributes and hence the engine and the arm should be in the same composite. Moreover, When the temperature of the devices reaches a provided critical value, all the requests to load or unload a container must be skipped and a message is sent to the supervisor. This requires to intercept calls on the provided interface of the controller to skip the controller action requests. As a result, the required view should encapsulate the controller, the engine, and the arm in the same composite. The required view is depicted in figure 5.3.

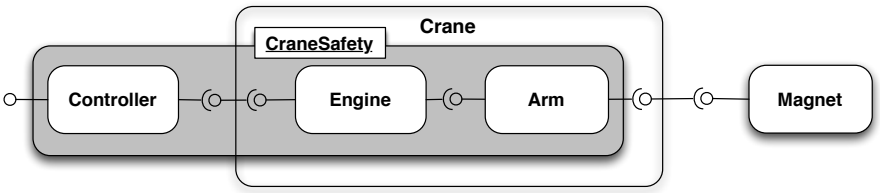

Figure 5.3: CraneSafety view of the crane

Implementing the real-time aspect

Real-time constraints can also be implemented as a wrapper on a crane system view. If the crane is in slow mode, then no real-time constraints are enforced. If the crane is in fast mode, then the time between the crane request to load/unload a container and setting the magnet off have to be achieved in t_{speed} time units. Otherwise, the arm should be moved up for its safety and all the subsequent user commands are refused and the supervisor must be warned. The required view in this case is the provided interface of the controller (to initialize the local time for the action when load or unload requests are intercepted), the required interface of the arm (to capture the interception time of setting the magnet off) and the provided interface of the arm (to call the arm to move up when the real-time constraint is violated). The corresponding view is shown in figure 5.4.

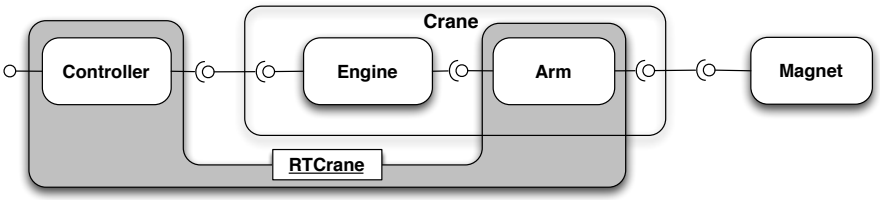

Figure 5.4: RTCrane view of the crane

As shown from the above examples, views make it simple to implement the above aspects: if the components are properly assembled (*i.e.,* the current configuration matches the required view), aspects are added as wrappers to the view otherwise a reconfiguration is needed. However, when we consider aspects requiring different views at the same time, wrappers (*i.e.,* aspects) crosscut each other as shown in Figure 5.5. It is obvious that the architecture of the component system must be reconfigured in order to enable different wrappers requiring different views. In the following, we introduce a specialized language for views definitions and show how it can be used in order to weave crosscutting wrappers.

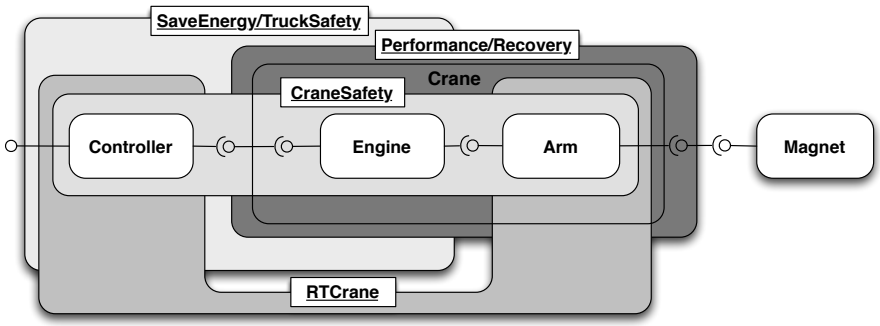

Figure 5.5: Wrappers crosscutting phenomenon

5.2 Views definition language

In this section we define a declarative language we call VIL (VIews Language) for the definition of views in component architectures. In the AOP point of view, VIL is a declarative pointcut language suitable for component models. Using VIL, minimum language constructs are used to define the points where aspects should interact with the system and this prevents the knowledge of the complete and detailed structure of the component systems. Here, we first define the join point model, then we introduce the complete syntax of the language and we describe its semantic.

5.2.1 The join point Model

The main element of each aspect-oriented language is the join point model. The join point model defines well-defined points in the execution of programs. These points are captured at runtime and aspects get involved only on those points [Kiczales 2001a]. On the other hand, components can be seen as black, gray or white boxes. Black box components are characterized with no specific knowledge of their internal behaviors, no access to their source code, and no knowledge of their architecture. In that case, the only accessible information of a component are its provided and required interfaces. Gray box components may show their internal architecture and potentially some knowledge of their behavior in terms of protocols or contracts [Szyperski 2002]. In particular, Fractal [Bruneton 2004] is a hierarchical component model where the architecture of composite components are given. Finally, white box components provides full access to their structure and behavior (*i.e.,* source code available). Thus, each of the above different categories of components requires an appropriate join point model when aspect-orientation is required. In our approach, we focus on the intermediate level (gray box component models) where components, their architectures and their behaviors specification are provided. While component architecture description enables the definition of views and the exact points where aspects are involved in the interaction with the base system, bahavior's specification enables aspect interferences detection. In our proposal, we consider calling services from required or provided services, and accessing component attributes. Inspired from

AspectJ [Kiczales 2001b], each service call at runtime is a different join point, and each component attribute access is a different join point. A join point has one of the forms `(cId, itfId, svId)` (for service call) and `(cId, atId)` (for attribute access) where `cId` ranks over component names, `itfId` and `atId` rank over interface and attribute identifiers, respectively, and `svId` ranks over service signatures. For example, the `(engine, iArm, moveUp(Speed))` denotes calling the `moveUp(Speed)` service of the `iArm` interface of the `engine` component, while `(engine, speed)` denotes the access to the attribute named `speed` of the `engine` component. The following section gives the complete syntax of VIL.

5.2.2 Syntax of VIL

In this section we describe our declarative pointcut language VIL for component models.

$$
\begin{array}{lcl}
vexp \in View & ::= & * \\
& | & cId \\
& | & \textbf{child}\ [^{+}]\ v \\
& | & \textbf{parent}\ [^{+}]\ v \\
& | & \textbf{primitive}\ v \\
& | & \textbf{instance}\ v \\
& | & [\textbf{direct}]\ \textbf{provide}\ v\ [(\textbf{T} \mid \textbf{N})\ id^{*}][sig^{*}] \\
& | & [\textbf{direct}]\ \textbf{require}\ \ v\ [(\textbf{T} \mid \textbf{N})\ id^{*}][sig^{*}] \\
& | & \textbf{bound}\ [\textbf{C} \mid \textbf{S}]\ v\ [id^{*}] \\
& | & \textbf{attributes}\ v\ [atId^{*}] \\
& | & \textbf{scflow}\ v \\
& | & v_1 \oplus v_2 \\
& | & v_1 \otimes v_2 \\
& | & v_1 \ominus v_2
\end{array}
$$

VIL defines a view in a component architecture specifying the set of components to be encapsulated together and their interfaces and services to be intercepted. In particular, "*" denotes that all the components of the architecture should be encapsulated in one component and all their interfaces and services are intercepted. Thus, the required view is the same as the original one (*i.e.,* no need to reconfigure the system) and the wrapper is to be plugged into the root component of the architecture. As described above, `cId` refers to a component identifier, this indicates that the required view is the same as the original one and all the interfaces of `cId` are intercepted, but in this case the wrapper is to be plugged into the component `cId`. **child** v and **parent** v denote that the view should encapsulate all the inner components (resp. parents) of v in the same component and all their interfaces are intercepted. "+" denotes a recursive closure of the above relations. **primitive** v denotes a view where all the primitive components of v should be in the same composite and all their interfaces are intercepted. **instance** v denotes a view where all the instances of v should be wrapped and all their interfaces are intercepted. **provide** v and **require** v denote the same view as v however, only the provided (rep. required) interfaces are intercepted. For more exhaustive specification we denote name (**N**) or type (**T**) patterns of the interfaces to be intercepted and potentially

the service signatures to be captured (*sig*). **bound** v denotes all the bound components to v. To be more precise we provide **C** and **S** to denote only the components bound to a client or to a server interfaces of v, respectively. An identifier pattern can also be used to select only the components whose names match a pattern identifier. **attributes** v denotes a view that encapsulates all the components in v and intercept accesses to all or some of their attributes. We also use **scflow** v to define a view that encapsulates all the components of v in the static control flow of the join points defined by v. Finally, a view can be composed of two other views using a union ($\oplus$), an intersection ($\otimes$) or a difference ($\ominus$) operators over views.

5.2.3 Semantics of VIL

Component architectures can alternately be represented as directed labeled graphs. Nodes in those graphs are the main architectural elements of the component architectures (*i.e., components, attributes, interfaces and services*), and the arc's labels denote the relationships among such elements (*e.g., child, provides, requires, boundTo*). These representation enables the introspection of component architectures and determine the location of elements of interest using an abstract language. FPath [David 2009b] is a domain specific language introduced to introspect Fractal architectures. FPath is a general and an extendible language and hence can be generalized for different component models by introducing new architectural elements and new relationships among elements. In the following we overview FPath and we show how our VIL expressions are evaluated using such a language.

5.2.3.1 FPath Query Language

FPath [David 2009b] is a query language developed to deal with the introspection of Fractal component architectures. FPath uses declarative path expressions to introspect Fractal elements: components, interfaces and attributes. FPath is jointly used with a scripting language named FScript to define complex runtime reconfiguration of Fractal architectures rather than using Fractal API which is a tedious and error prone task. FPath expressions are of the following form, where *nodeId* is a node identifier denoting the starting point from which the navigation starts and *arcId* denotes a transition label identifier denoting the axis to follow:

$$\begin{array}{lll} \mathit{fExp} & ::= & \$\mathit{nodeId}\ \text{“/”}\ \mathit{step}\ (\text{“/”}\ \mathit{step})^* \\ \mathit{step} & ::= & \mathit{arcId}\ \text{“::”}\ (\mathit{nodeId} \mid \star)\ [\mathit{predicate(.)}] \end{array}$$

For example, the FPath expression "*$crane/child::*/requires::**" returns the set of required interfaces (the arc label $requires :: *$) of all the inner components ($child :: *$) of the crane composite component (*$crane*). However, the current version of FPath cannot directly be used in our proposal for several reasons: (1) FPath expressions return primitive elements: a set of components or interfaces or attributes while in our join point model we need to define a set of tuples defining the interface services of components to be intercepted. (2) FPath does not consider

low level architectural elements such as services. (3) some of the current relationships are related to Fractal (*e.g.,* internal interfaces) where we need an independent language. For the above mentioned reasons, we adapt FPath to fulfil our requirements. Figure 5.6 depicts the architectural elements and the relationships required by our model. In the figure, the architectural elements considered are depicted with named circles: *component*, *interface*, *service* and *attribute*. The relationships among the elements are represented by labled arrows: *parent*, *child*, *provides*, *requires*, *has*, *boundTo*, *defines*, *set* and *get*. The elements and the relationships annotated with $\star$ are the ones newly added to FPath.

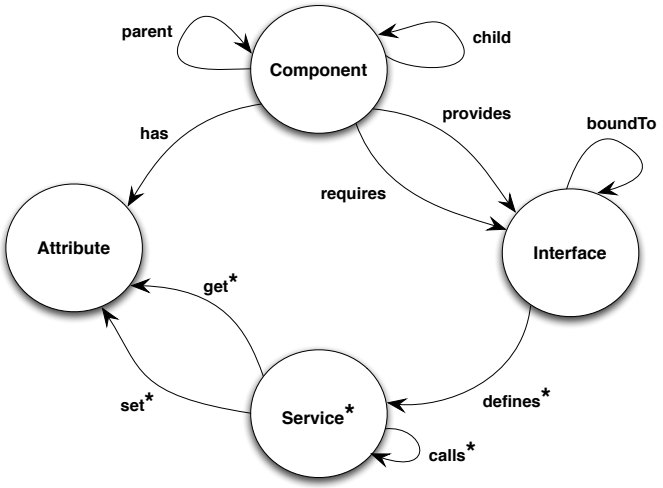

Figure 5.6: Directed labeled graph adopted for component architectures

5.2.3.2 VIL semantics in FPath

In this section we give an excerpt of the semantics of VIL, the rest can be straightforwardly deduced. VIL uses FPath to access the different architectural elements on a component architecture, and defines the different join points required to be intercepted. In addition, a views transformation function is defined for VIL to generate the required views (*i.e.,* a view where all the components in the jon points are wrapped). First we introduce a function *eval* that defines the set of join points from VIL expressions for a component architecture a. Let us consider the primitive expression cId, the *eval* function calls FPath ($\mathcal{F}_{path}$ function) to get all the interfaces of cId (*e.g., itfs*) and all the services defined for each interface (*e.g.,* svs_{itf}), finally it defines the set of join points as a set tuples $\{(cId, itf, sv) : itf \in itfs \wedge sv \in svs_{itf}\}$. Formally:

$$eval\ (cId)\ a \quad = \quad \textbf{let}\ e_1 = \$\texttt{cId/provides} :: *,\ e_2 = \$\texttt{cId/requires} :: *,\ itfs = \mathcal{F}_{path}[\![e_1]\!]\ a \cup \mathcal{F}_{path}[\![e_2]\!]\ a\ \textbf{in} \bigcup_{\forall itf \in itfs} \{(cId, itf, s) :\ s \in \mathcal{F}_{path}[\![(\texttt{e}_1|\texttt{e}_2)[\texttt{name}(.) = \texttt{itf}]/\texttt{service} :: *]\!]\ a\}$$

Another example is the expression (**child** v) which is evaluated by evaluating v first and for each component in v we get all its direct inner components capturing all their services of all their provided and required interfaces. The $s_{\downarrow_t}$ in the formula denotes a projection operation that extracts only the t elements from the tuples defined by s.

$$eval\ (\textbf{child}\ v)\ a \quad = \quad \textbf{let}\ cs = (eval\ v\ a)_{\downarrow component}\ \textbf{in} \bigcup_{\forall cId' \in (\forall cId \in cs : \mathcal{F}_{path}[\![\$\texttt{cId/child}::*]\!])} eval\ cId'\ a$$

The expression (**attributes** v) is evaluated by applying the *eval* function to v capturing all the attributes defined on all the components of v.

$$eval\ (\textbf{attributes}\ v)\ a \quad = \quad \textbf{let}\ cs = (eval\ v\ a)_{\downarrow component}\ \textbf{in} \bigcup_{\forall cId \in\ cs} \{(cId, at) :\ at \in \mathcal{F}_{path}[\![\$\texttt{cId/has} :: *]\!]\ a\}$$

A more interesting example is the (**scflow** v) which is evaluated by evaluating v capturing the set of join points on the recursive closure of `calls` relation of each service on v.

$$eval\ (\textbf{scflow}\ v)\ a \quad = \quad \textbf{let}\ jps = eval\ v\ a\ \textbf{in} \bigcup_{\forall sv \in\ jps_{\downarrow service}} \{(cId, itf, s) :\ s \in \mathcal{F}_{path}[\![\$\texttt{sv/calls}^+ :: *]\!]\ a\}$$

A last example is views composition operations that forms one view from two predefined views. Here is the definition of the union operation of views, that constructs a new view that encapsulates all the components and intercepts all the join points encapsulated and intercepted by its underlying views.

$$eval\ (v_1\ \oplus\ v_2)\ a \quad = \quad eval\ v_1\ a\ \cup\ eval\ v_2\ a$$

In addition to the above *eval* function, we provide views transformation function σ_{mId}. This function is a model-independent function. It defines how wrappers and

views are represented according to the basic tenets of a component model mId. In the following, we show how σ_{mId} is defined to model views and wrappers for two different categories of component models: hierarchical component models with component sharing such as Fractal and flat component models such as EJB.

5.3 Implementation of VIL in Fractal component model

Fractal component model [Bruneton 2004] defines its own architecture description language (ADL) to describe component assemblies. It supports hierarchies, component controllers, introspection and component sharing. Fractal controllers intercept calls to a component provided and/or required interfaces and enable component behavior adaptation and the definition of extra-functional features to components by executing an additional code when calls are intercepted. We get benefit of Fractal controllers to define aspects (*i.e.,* wrappers). However, in Fractal Julia implementation, when a component has several controllers, there is no general way to compose them. They can only be executed independently or sequentially by configuring interceptors or be composed in a programmatic way by explicitly calling one controller from another. This makes the implementation of controller-based adaptations a complex task with sometimes unexpected behavior. To tackle this limitation, we introduce composable controllers into Fractal Julia implementation. Composable controllers, as their name indicate, get benefits of regular Fractal controllers (control the behavior of components by introducing extra-functional properties) and enable controller composition. In the following, we introduce the concept of composable controllers, we describe how views are defined using VIL, and we show how aspects (*i.e.,* controllers) can be composed to solve potential interferences.

5.3.1 Composable controllers

We define a composable controller as a pair `(Dispatcher,ICController)` where `Dispatcher` is a regular Fractal controller plugged into the composite defining the view and `ICController` is an object implementing the `ICController` interface (see Listing 5.1).

```
1 enum Action {Proceed, Skip}
2 interface ICController {
3   Action match(MessageContext c);
4 }
```

Listing 5.1: `ICController` interface

The `Dispatcher` controller intercepts calls to the inner components of the view, reifies the intercepted calls into `MessageContext` objects and calls the `match()` method of the `ICController`. The `Dispatcher` waits for the action taken by the `ICController` (*i.e.,* `Proceed` or `Skip`) and behaves accordingly. When it receives a `Proceed` action, it calls the original method and the call reaches its target, when it

receives a `Skip` the `Dispatcher` does nothing and the call is ignored. For composing controllers, composition operators are defined as `ICController`(s), this enables the composition of controllers and composition operators in a composite pattern way. Each `match(MessageContext)` method of the composition operator implements the semantics associated to each operator and returns `Proceed` or `Skip` accordingly to the `Dispatcher` controller. Take for example the `crane` view of the crane system (see Figure 5.7).

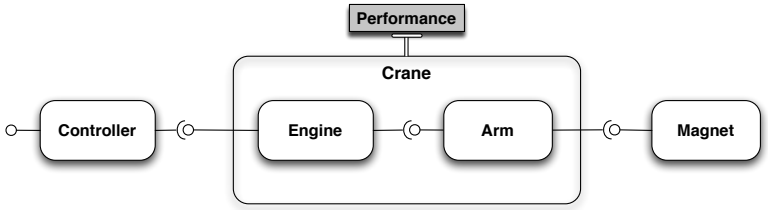

Figure 5.7: A composable controller on the `Performance` view of the crane where the `ICController` is shown as a gray box with the name of the aspect and the `Dispatcher` is depicted with (⊤) at the top of the view

The integration of the `Performance` aspect to the crane requires adding a composbale controller `(Dispatcher,Performance)` to the original view. The `Dispatcher`, in this case, intercepts the incoming calls to the `Engine` and the outgoing calls from the `Arm`. For each intercepted call, the `Dispatcher` reifies it and calls the `match()` method of the `Performance` object (see Listing 5.2). The `match()` method of the `Performance` checks whether the intercepted call is a `setOn()` (resp. `setOf()`) method call (line 6-7). If it is the case, it stores the state of the magnet in the attribute `isMagnetOn`. If the intercepted call is a `moveLeft(Speed)` or a `moveRight(Speed)` method call (line 8), it first checks the state of the magnet, if it is off, it changes the parameter value of the call to `fast` by calling `c.setSpeedArgument("fast")` method (code not shown here). Finally, the `match()` method returns `Proceed` (line 11) thus the `Dispatcher` calls the original method which reaches its target.

```
1 class Performance implements ICController {
2
3   private boolean isMagnetOn = false;
4
5   Action match(MessageContext c) {
6     if (c.getSignature().equals("setOn()")) isMagnetOn = true;
7     else if (c.getSignature().equals("setOff()")) isMagnetOn = false;
8       else if (c.getSignature().startsWith("move")) {
9           if (isMagnetOn()) c.setSpeedArgument("fast");
10      }
11    return Action.Proceed;
12  }
13 }
```

Listing 5.2: The `Performance` implementation

Now, we want to apply the `TruckSafety` and the `SaveEnergy` aspects, both require another view where the `Controller` and the `Engine` to be in the same component. However, since both aspects intercept the same join points, a composition

strategy should be used. In our approach we adopt the sequential composition as a default strategy for aspects sharing join points. The sequential composition is abstracted with a generic and a reusable operator named `Seq`. The `Seq` operator forwards calls to shared join points to both aspects in a sequential order and proceeds a call only when at least one of its underlying aspects returns `Proceed`, otherwise it skips the call (see Chapter 6 for the full semantics of operators). Figure 5.8 depicts the Fractal architecture of the crane example after applying both the `TruckSafety` and the `SaveEnergy` aspects (*i.e.,* `ICController`(s)). In the figure, the new view crosscuts the original view, thanks to component sharing, this configuration is possible (*i.e.,* the `Engine` component is a shared component between the two views). Shared components are depicted with dashed lines in the figure.

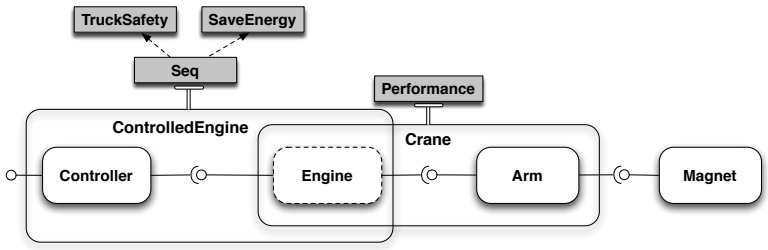

Figure 5.8: `Seq(TruckSafety,SaveEnergy)` plugged into `ControlledEngine` view

Let us generalize and detail our implementation of views for Fractal. For the definition of views in Fractal Julia, we distinguish two cases: (1) the components to be wrapped belong to the same composite, and (2) the components to be wrapped do not belong to the same composite. In the following we detail how views are created for each case.

5.3.2 The components of interest belong to the same composite

In this case, the required view is the same as the original configuration. In practice, this requires to update the ADL description of the architecture by (1) declaring a composable controller modelling the required aspect as a part of the membrane of the composite encapsulating the components of interest and (2) by declaring an interceptor controller as a part of the membrane of each component of interest. In addition, a Julia configuration file is generated to include the definition of the added controllers. Listing 5.3 shows the general required modification of the ADL description where the component named *c* models the required view by a VIL expression. In that case, a composable controller is defined for that component (`CControllerName` line 12), and an interceptor controller is defined for each one of its inner components (`Interceptor` line 10). Each interceptor controller intercepts incoming and/or outgoing calls to/from its underlying component and forwards them to the `Dispatcher` of the composable controller of its component parent.

```
1 <component name="c">
2     // interface declarations
```

```
3     // the content declaration if "c" is a primitive
4     // else for each inner component ci
5     <component name= "ci">
6       // interface declarations of the inner component
7       // the content declaration if "ci" is a primitive
8       // or the inner components declaration if "ci" is a composite
9       // binding declarations if "ci" is a composite
10      <controller desc = "Interceptor"/>
11    </component>
12  <controller desc = "CControllerName"/>
13 </component>
```

Listing 5.3: The required ADL modification if the view corresponds to the original configuration: the underlined code is the one that is added to model the wrapper

In addition to the above ADL description modification, a Julia configuration file should be defined to include the full description of each controller: its implementation class, the required interceptors and a controller composition expression if any. Listing 5.4 shows the structure of the Julia configuration file. The file starts with an indication of the composable controller identifier, the same used in the ADL description (*i.e.,* `CControllerName` line 1). The interface of the controller is defined (`ICController` line 4). The implementation class of the controller is defined by the `Dispatcher` class name followed by *ccExp* expression that is passed as a parameter to the `Dispatcher` class. The *ccExp* expression is of the form:

$$\begin{array}{lcl} \langle ccExp \rangle & ::= & ICControllerName \mid \langle opId \rangle\ (\langle ccExp \rangle, \langle ccExp \rangle) \\ \langle opId \rangle & ::= & Seq \mid Fst \mid Alt \mid ... \end{array}$$

In addition, the `InterceptorKind` is one of three kinds: `InInterceptor` that intercepts only the incoming calls of a component, `OutInterceptor` that intercepts only the outgoing service calls of a component, and `InOutInterceptor` that intercepts all the incoming and the outgoing calls of a component.

```
1 (CControllerName
2   (
3     'interface-class-generator
4     (
5       (CControllerName-controller vil.common.ICController)
6     )
7     (
8       (vil.controllers.Dispatcher <ccExp>)
9     )
10    (
11      (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
12        org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
13      )
14    )
15    org.objectweb.fractal.julia.asm.MergeClassGenerator
16    'optimizationLevel
17  )
18 )
19
20 (Interceptor
21   (
22     'interface-class-generator
23     (
```

```
24        (Interceptor-controller vil.common.IInterceptor)
25      )
26      (
27        (vil.controllers.Interceptor)
28      )
29      (
30        (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
31          org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
32          (vil.interceptors.InterceptorKind 'interfaceName)
33        )
34      )
35      org.objectweb.fractal.julia.asm.MergeClassGenerator
36      'optimizationLevel
37    )
38 )
```

Listing 5.4: The generated Julia configuration file for each view definition

For better understanding of our modelization, take for example the crane example with its crane view. Listing 5.5 shows the updated ADL description of the crane example after adding the `Performance` aspect to the crane component. The only modification is the addition of interceptor controllers `Interceptor` to the inner components of the `Crane` (line 15,21) and a composable controller named `Performance` to the `Crane` composite (line 26).

```
1 <component name=root>
2   <interface name="iController" signature="Crane.IController" role="server">
3   <component name="controller">
4     <interface name="iController" signature="Crane.IController" role="server"
          >
5     <interface name="iEngine" signature="Crane.IEngine" role="client">
6     <content class="Crane.ControllerImpl">
7   </component>
8   <component name="crane">
9     <interface name="iEngine" signature="Crane.IEngine" role="server">
10    <interface name="iMagnet" signature="Crane.IMagnet" role="client">
11    <component name=engine>
12      <interface name="iEngine" signature="Crane.IEngine" role="server">
13      <interface name="iArm" signature="Crane.IArm" role="client">
14      <content class="Crane.EngineImpl">
15      <controller desc = Interceptor/>
16    </component>
17    <component name="arm">
18      <interface name="iArm" signature="Crane.IArm" role="server">
19      <interface name="iMagnet" signature="Crane.IMagnet" role="client">
20      <content class="Crane.ArmImpl">
21      <controller desc = Interceptor/>
22    </component>
23    <binding client="this.iEngine" server="engine.iEngine"/>
24    <binding client="engine.iArm" server="arm.iArm"/>
25    <binding client="arm.iMagnet" server="this.iMagnet"/>
26    <controller desc = Performance/>
27  </component>
28  <component name="magnet">
29    <interface name="iMagnet" signature="Crane.IMagnet" role="server">
30    <content class="Crane.MagnetImpl">
31  </component>
32  <binding client="this.iController" server="controller.iController"/>
33  <binding client="controller.iEngine" server="crane.iEngine"/>
34  <binding client="crane.iMagnet" server="magnet.iMagnet"/>
```

```
35 </component>
```

Listing 5.5: The Fractal ADL description of the crane example after adding the `Performance` aspect

Listing 5.6 describes the Julia configuration file that defines the `Performance` controller. In the listing, the interface implemented by the controller (*i.e.,* `ICController`) is indicated at line 5. The `Dispatcher` controller is indicated at line 8 where the `Performance` is passed as a parameter to the `Dispatcher`. Thus, an object of the class `Performance` is created for the `Dispatcher` controller. The rest of the code in the listing indicates how the code of the controller is merged with the component implementation code.

```
1 (Performance
2   (
3     'interface-class-generator
4     (
5       (performance-controller vil.common.ICController)
6     )
7     (
8       (vil.controllers.Dispatcher Performance)
9     )
10    (
11      (org.objectweb.fractal.julia.asm.InterceptorClassGenerator
12        org.objectweb.fractal.julia.asm.LifeCycleCodeGenerator
13      )
14    )
15    org.objectweb.fractal.julia.asm.MergeClassGenerator
16    'optimizationLevel
17  )
18 )
```

Listing 5.6: The Julia configuration file for the `Performance` aspect

5.3.3 The components of interest are scattered in the architecture

Here we need to reconfigure the system to accept the required view. Thanks to component sharing feature in Fractal, this kind of reconfiguration is possible. In Fractal we adopt the following reconfiguration strategy: a new composite is created as a child of the closest common parent of the required components. This composite shares all the required components with their original parents. Similar to the above case, the composable controller is plugged into the new composite and an interceptor controller is added to the membrane of each inner component. Thus, the view is added and the original configuration is preserved. The choice for the position of the new composite declaration is made to synchronize the life-cycle of the view with the life-cycle of its inner components. Thus, when all the inner components of the view are destroyed, the view is automatically destroyed. In addition, when two views crosscut each other and an interference appears between their composable controller, both views are composed into one view using the union operator ($\oplus$). This enables

the composition of the composable controllers of different views to solve potential interferences among them.

Thus, the definition of a view in this case is divided into two steps. The first step consists in finding the closest common parent of the components to be wrapped. This can be done using FPath language: consider c_{11} and c_{21} two different components belonging to c_1 and c_2, respectively. The following FPath expression provides a set of all their common parents including the root component:

$$e = c_{11}/ancestor :: *[in(c_{21}/ancestor :: *)]$$

The "$c_{11}/ancestor :: *$" sub-expression returns the set of all the ancestors of c_{11} including the root component. With the predicate "*in*" presented between square brackets, only the ancestors of c_{11} that belong to the set of ancestors of c_{21} will be returned. The closest parent c belongs to that set and has the following particularity: $descendant(c) \cap \mathcal{F}_{path}(e) = \phi$ which states that none of the descendants of the closest parent is a common parent of the required components. The second step consists in adding a new composite as an inner component of the common parent of c_{11} and c_{21} found by the previous step. This is shown in Listing 5.7. The new composite *view* (line 24-28) declares c_{11} and c_{21} as its inner components sharing them with their original parents c_1 and c_2, respectively (line 25-26). This way, the original architecture is preserved after the new view is defined. Finally, an interceptor controller is associated to each shared component (line 9,19), and a composable controller is added to the new created composite (line 27). The added interceptor controllers intercept calls on their components and route them to the nesting composite.

```
1  <component name="c1">
2      // interface declarations
3      <component name= "c11">
4        // interface declarations of the inner component
5        // the content declaration if "c11" is a primitive
6        // or the inner components declaration if "c11" is a composite
7        // binding declarations if "c11" is a composite
8        <controller desc = "Interceptor"/>
9      </component>
10     // other inner components of c1
11 </component>
12 <component name="c2">
13     // interface declarations
14     <component name= "c21">
15       // interface declarations of the inner component
16       // the content declaration if "c21" is a primitive
17       // or the inner components declaration if "c21" is a composite
18       // binding declarations if "c21" is a composite
19       <controller desc = "Interceptor"/>
20     </component>
21     // other inner components of c2
22 </component>
23 // other component declarations of this level
24 <component name = "view">
25     <component name = "c11" definition = "/c1/c11"/>
26     <component name = "c21" definition = "/c2/c21"/>
```

```
27     <controller desc = "CControllerName"/>
28 </component>
```

Listing 5.7: The required ADL modification if the components of interest are scattered in the architecture

Listing 5.8 shows the ADL description of the reconfigured crane example to include the composition of the `truckSafety` and the `saveEnergy` aspects. The added lines are underlined in the script. The Julia configuration file generated for the view includes the definition given in Listing 5.4 replacing $\langle ccExp \rangle$ with `Seq(truckSafety,saveEnergy)`.

```
1 <component name=root>
2   <interface name="iController" signature="Crane.IController" role="server">
3   <component name="controller">
4     <interface name="iController" signature="Crane.IController" role="server"
          >
5     <interface name="iEngine" signature="Crane.IEngine" role="client">
6     <content class="Crane.ControllerImpl">
7     <controller desc = Interceptor/>
8   </component>
9   <component name="crane">
10    <interface name="iEngine" signature="Crane.IEngine" role="server">
11    <interface name="iMagnet" signature="Crane.IMagnet" role="client">
12    <component name=engine>
13      <interface name="iEngine" signature="Crane.IEngine" role="server">
14      <interface name="iArm" signature="Crane.IArm" role="client">
15      <content class="Crane.EngineImpl">
16      <controller desc = Interceptor/>
17    </component>
18    <component name="arm">
19      <interface name="iArm" signature="Crane.IArm" role="server">
20      <interface name="iMagnet" signature="Crane.IMagnet" role="client">
21      <content class="Crane.ArmImpl">
22    </component>
23    <binding client="this.iEngine" server="engine.iEngine"/>
24    <binding client="engine.iArm" server="arm.iArm"/>
25    <binding client="arm.iMagnet" server="this.iMagnet"/>
26  </component>
27  <component name="magnet">
28    <interface name="iMagnet" signature="Crane.IMagnet" role="server">
29    <content class="Crane.MagnetImpl">
30  </component>
31  <component name = "ControlledEngine">
32    <component name = "controller" definition = "/controller"/>
33    <component name = "engine" definition = "/crane/engine"/>
34    <controller desc = "Seq - truckSafety - saveEnergy"/>
35  </component>
36  <binding client="this.iController" server="controller.iController"/>
37  <binding client="controller.iEngine" server="crane.iEngine"/>
38  <binding client="crane.iMagnet" server="magnet.iMagnet"/>
39 </component>
```

Listing 5.8: The Fractal ADL description of the crane example after adding the `Seq(TruckSafety,SaveEnergy)` composable controller

5.3.4 Fractal Weaver

The above component architecture transformation becomes a tedious and error prone task when the architecture grows. Our approach makes it possible to automatize this task. For that reason we developed a Fractal weaver as a top level extension of Fractal Julia implementation as shown in Figure 5.9. In the figure, the Fractal weaver consists of three modules: *VIL analyzer*, *ADL transformer*, and *Julia config generator*. In the following we describe the role of each module.

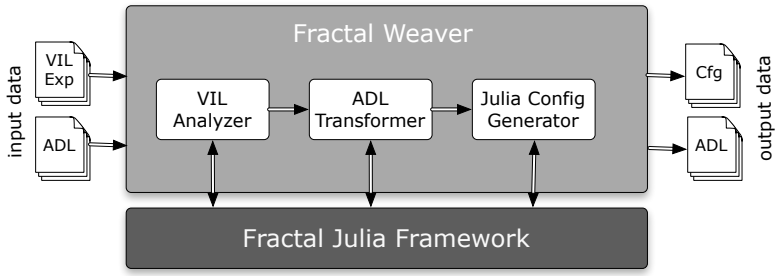

Figure 5.9: Fractal Weaver Architecture

5.3.4.1 VIL Analyzer

This module analyzes the input VIL expressions, and uses Julia introspection mechanism to introspect the given ADL description and returns the concrete join points to be intercepted. When an expression is not well-defined, an exception is thrown indicating an error in the input expression, otherwise a set of concrete join points is returned (see Section 5.2.1). Figure 5.10 describes the UML class diagram associated to VIL expressions.

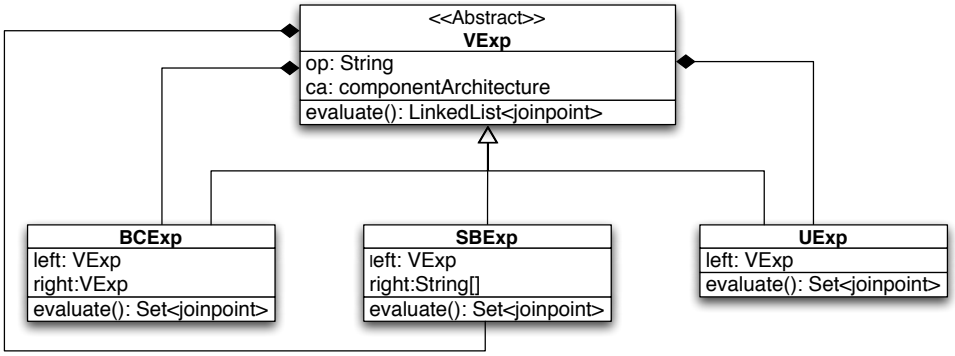

Figure 5.10: VIL Expressions structure

A VIL expression is of three forms: (1) Unary form (`UExp`) where the expression has an operator and merely one operand. This includes **child**, **parent** and **instance** expressions. (2) Selective binary form (`SBExp`) where the right hand side operand is a set of patterns or selectors that match a subset of the find join points. This includes

provide, **require** and **direct** expressions. (3) Binary composition of expressions (`BCExp`) where views are composed using $\oplus$, $\ominus$ and $\otimes$ operators. The analyzer role is to interpret VIL expressions and returns a set of join points (`Set<joinpoint>`). The `evaluate()` function of each class implements the required behavior of the analyzer for the form in question.

5.3.4.2 ADL Transformer

This module uses the returned join points from the above module and uses them to determine and define the required views. It uses the process described in Section 5.3.2 and Section 5.3.3, respectively. This includes the addition of new composites, the declaration of composable controllers and interceptor controllers for components, and the composition views when needed (see Section 5.3.3).

```
1 interface IADLTransformer {
2   Dag Adl2Dag(File src);
3   File Dag2Adl(Dag dag);
4   run(ComponentArchitecture ca, String asId, VExp exp);
5 }
```

Listing 5.9: ADL Transformer Interface

The ADL transformer uses an intermediate DAG structure to transform the ADL description into a component architecture with the required views and controllers. Listing 5.9 describes the interface implemented by the ADL transformer. It defines three methods: `Adl2Dag()` and `Dag2Adl()` to transform an ADL description into a DAG structure and vice versa, and a `run()` method that calls the `Adl2Dag()` method to get the DAG structure of the original architecture, then it calls the VIL analyzer to interpret a VIL expression, update the DAG structure to consider the required view and calls the `Dag2Adl()` method to get the final ADL description with the required views and composable controller(s).

5.3.4.3 Julia Config Generator

This latter module generates the required Julia configuration files providing information about the implementation of composable controllers and interceptors. These files are required for the correct execution of the result system in Julia. It automatizes the step described in Section 5.3.3.

5.4 Implementation of VIL in EJB component model

For flat component models, such as EJB, a view cannot be modeled as a composite since component hierarchy is not supported by such models. In addition, neither controllers nor component sharing are supported. One solution is to model a wrapper as a pair of regular components or beans `(Dispacher,Aspect)`. The `Dispatcher` bean is bound to all the beans of interest on the interfaces to be intercepted. Since the `Dispatcher` in this case has no a general structure (*i.e.,* no predefined number and type of interfaces), it can be automatically generated from the join points

definition (*i.e.,* defines all the interfaces in the set of defined join points). Like the one in Fractal, the dispatcher intercepts all the calls, defined by VIL expressions, instead of their original targets, reifies and forwards the calls to the aspect bean. The `Aspect` bean executes extra-code and decides whether to proceed or skip the call by returning `proceed` or `skip` to the dispatcher. This latter proceeds calls by calling the original service and skips them by ignoring them. Implementing this solution, the component system is reconfigured by adding the components modelling the dispatcher and the aspect bean to the container and update the bindings to fulfil the required view as depicted in Figure 5.11. In the figure, the dispatcher and

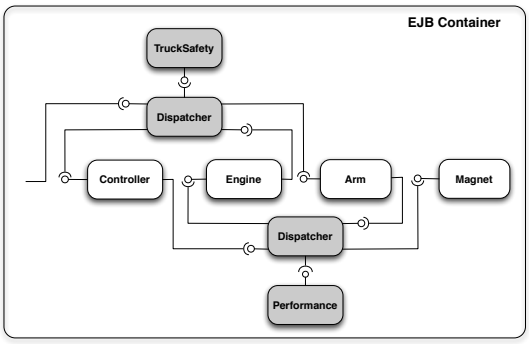

Figure 5.11: The implementation of views and wrappers in flat component models

the aspect beans are depicted with gray boxes. Two crosscutting views (`Crane` and the `ControlledEngine`) are presented. The `Dispatcher` bean in this case defines two required interfaces (*i.e.,* `IController` and `IArm`) and two provided interfaces (*i.e.,* `IController` and `IArm`). An `Aspect` bean always provides only one interface similar to that defined for Fractal (*i.e.,* `ICController` interface). This enables the composition of aspect beans using composition operator. A composition operator can also be modeled as components or beans that intercept reified join points and forward them to beans modelling the different aspects following their appropriate strategy, then return the result to the dispatcher. Figure 5.12 shows how the `TruckSafety` and the `SaveEnergy` wrappers are composed using the `Seq` operator.

5.5 Conclusion

In this chapter, we described our approach based on views and wrappers for aspectualizing component models. Views can be defined using VIL language in a declarative style. VIL is a pointcut language adopted for component models. Crosscutting views can be modeled in different ways following the basic tenets of each component model. Our approach is designed to be model and language-independent, we have shown how it can be used for hierarchical component models with component sharing such as Fractal and flat component models such as EJB. Currently, the approach is implemented for Fractal component model and the Fractal weaver presented in Section 5.3.4 is available upon request.

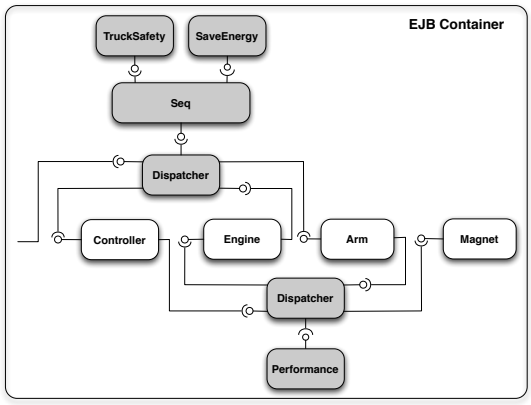

Figure 5.12: Wrappers composition in flat component models

Chapter 6

Aspects Interferences Detection and Resolution

Contents

In this chapter, we show how aspect interferences can be detected and potentially solved using model checking approaches and composition operators, respectively. In our proposal we adopt the use of Uppaal model checker for modelling component systems, aspects, and aspectualized component systems, for formal detection of aspect-base and aspect-aspect interferences. First, we give a short overview of Uppaal. Second, we describe a transformation scheme of component systems into

Uppaal processes. Third, we show how base system and aspect properties can be specified in CTL formulas and checked within Uppaal to detect interferences. Finally, we present a set of composition operators as patterns for aspect interference resolution.

6.1 Overview of Uppaal

Uppaal [Bengtsson 1996, Larsen 1997, Behrmann 2004] is a toolbox used to design, simulate and check properties for systems that can be modelled as state machines extended with local variables, data types, and clock variables. Such kind of state machines with time support are called timed automata [Alur 1992]. For modelling time, Uppaal uses a dense time logic [Ahmed 1996] where clock variables range over real numbers, but they can be reset and assigned to natural numbers. All the clocks of a system start at the same instant and they proceed at the same rate and hence clocks of the same system progress synchronously. Each state machine in Uppaal is called a *template*. A template can be parametrized with constants and data variables indicating how that template is instantiated (*e.g.,* how many instances will be created). Each instance is called a *process*. Template nodes are called *locations* while the edges are called *transitions*. The Uppaal toolbox consists of three parts: a modelling description language, a simulator and a model checker. In the following we give a brief description of each part of Uppaal with an abstract client-server example.

6.1.1 Description language

For each Uppaal template, edges are decorated with *channels*, *guards* and *reset operations*. Channels are the communication means between processes in Uppaal. Thus, Uppaal uses `a!` and `a?` to denote sending and receiving a channel `a`, respectively. In addition, channels can be broadcasted to several processes, where different processes can synchronize with the same channel. Guards express conditions on data variables and clocks that must be satisfied to enable transitions. Reset operations are sequences of data variables and clocks assignment that can also be expressed as C functions in the declaration part of templates. Channels, guards and reset operations presence is optional in transitions. In particular, the absence of a channel indicates an internal action of the process. To enforce timing constraints, Uppaal supports invariants on locations to indicate that the system cannot remain in a particular location more than a specified time value associated to a clock variable. In addition, Uppaal supports parameter passing between processes.

For better understanding of Uppaal modelling language, let us consider an abstract client-server example that we show in figure 6.1. In the figure, two templates are presented, one for a server (left hand side figure) and one for a client (right hand side figure). The initial location `S0` is indicated with a double circle. The transition (`S0 -> S1`) in the server template is decorated with a condition (*isActive()* in italic font), a received channel name (`a[id]?` in normal font) and a reset operation

(**update()** in bold font). The transition (`S0 -> S2`) is decorated only with a condition (*!isActive()*) and a receiving channel (`a[id]?`). That is to indicate that from an initial location, a server waits for a message `a[id]?`. The (`S0 -> S1`) transition is enabled if the server is active (*isActive()* evaluates to true). In that case, the reset operation **update()** is executed and the system goes to `S1` location. If the server is not active (*!isActive()* evaluates to false), the (`S0 -> S2`) transition is enabled and the system goes to the `S2` location. In both cases, the system is not allowed to stay at the new location more than 5 minutes (`cl<=5` variant), instead, the process returns to the initial location by either sending `b[ret]!` or `c!` channels to the corresponding client and resets the clock variable (`cl:=0`). Since there are more than one instance of the server (the server template is parametrized with `const ID id`, where `ID` is a data type of range `[0,N]` and `N` is a constant indicating the number of processes that should be instantiated for the template), a client arbitrary chooses one of the servers (`id:ID` selector in underlined font), sends a channel `a[id]!` to the server `id` and goes to the `S1` location. It stays at that location waiting for either `b[ret]?` or `c?` channels, when this happens, the client returns directly to the initial location. The initial location of the client is denoted with the symbol (`U`) to indicate that this location is urgent and hence the client does not stay at that location since there is a server enable to communicate with the client. Furthermore, the server process passes a returned value to the client in the `b[ret]!` channel, the returned value is saved in the local variable `val` (*i.e.,* `val:=ret`).

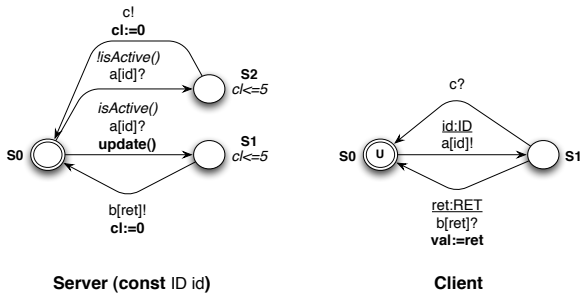

Figure 6.1: Uppaal graphical description of a client-server example

For describing systems, Uppaal provides both a graphical (XML) format and a textual (XTA) format. Within the graphical format, templates can be designed as graphs with nodes and edges following the WYSIWYV principle (*i.e.,* What You See Is What You Verify) as shown in figure 6.1. The textual format provides to the user a programming language of automata. Listing 6.1 describes the XTA description of the client server example. In the figure, constants, data types and channels are declared first (line 2-5), in the global declaration part followed by a set of processes declarations. Each process has an identifier and potential parameters. For example, the server process is parametrized with `const ID id` to denote that several instances (3 in this case) of the template are available. Then the set of local variables, clocks, and potential C functions used for guards or assignments

are declared. States and transitions are declared at the end of process declaration. Finally, the set of concurrent processes are indicated using the `system` clause.

```
1 // global declarations
2 const N = 3;
3 typedef int [1, N] ID;
4 typedef int [0,5] RET;
5 chan a[ID], b[RET], c;
6
7 process Server(const ID id) {
8 // local declarations
9 clock cl;
10 RET ret := 0;
11
12 bool isActive () {}
13 void update() {}
14 state S0, S1 {cl<=5}, S2 {cl<=5};
15 init S2;
16 trans
17     S0 -> S1 { guard isActive(); sync a[id]?; assign update();},
18     S0 -> S2 { guard !isActive(); sync a[id]?;},
19     S1 -> S0 { sync b[ret]!; assign cl:=0;},
20     S2 -> S0 { sync c!; assign cl:=0;};
21 }
22
23 process Client () {
24 RET val;
25 state S0, S1;
26 urgent S1;
27 init S1;
28 trans
29     S0 -> S1 { select id:ID; sync a[id]!;},
30     S1 -> S0 { sync c?;},
31     S1 -> S0 { select ret:RET; sync b[ret]?; assign val:=ret;};
32 }
33 // The list of processes to be composed into a system
34 system Client, Server;
```

Listing 6.1: Textual description (XTA) of the client server example

6.1.2 Simulator

Uppaal provides a simulator for exhaustive examination of systems behaviors. Within a simulator, a user can interact with the system, execute the system step by step, decide which transition should be taken when several are enabled, and see how the data variables and clocks values change during the execution. In addition, the simulator visualizes traces generated by the model checker.

6.1.3 Model checker

The Uppaal model checker is designed to check reachability, safety and liveness properties expressed in a subset of CTL (Computation Tree Logic) formulae. When a particular property is violated, a counter example in terms of a diagnosis trace is automatically reported to the user. Thus, the user is given information to detect potential errors and helped to correct them. Uppaal model checker supports two kinds of formulas: *state formulae* and *path formulae*. While the former evaluate the

system in individual states, the latter, focus on the model behavior over traces of its execution.

State formulae evaluate state variables in individual locations (*e.g.,* `val==5`), or checks whether a process is in a given location (*e.g.,* `Server[id].S1`). The deadlock freedom is a state formula that is checked for every state using a special keyword (`not deadlock`). State formulae can be combined using regular logical operators (*i.e.,* and &&, or ||, not `!`).

Path formulae are divided into reachability, safety, and liveness properties. Reachability properties in Uppaal express that *"some states satisfying a property are reachable"*, this is written as: `E<>` ϕ where ϕ is a state formula. Safety properties in Uppaal express that *"something good is invariantly true"*, these are written in Uppaal as: `A[]` ϕ (for all the paths and all the sates in each path) and `E[]` ϕ (for all the states of some paths). Liveness properties express that *"something will eventually happen"*, this is written as `A<>` ϕ or *"whenever a property is satisfied, another property is eventually satisfied"*, this is written as ϕ_1 `-->` ϕ_2.

In our proposal, the reason for the adoption of Uppaal model checker is twofold: (1) generality and (2) expressiveness. By generality, both real time systems and non real time systems can be modelled thanks to the use of clocks in the process templates. While by expressiveness, we get the benefit of state variable assignments, parameter passing between processes and templates instantiation which are intrinsic properties for components: components may have attributes that can be updated during their life cycles, they may exchange values and they can be instantiated several times in the same system.

6.2 Formalization of component systems in Uppaal

In this section we describe a transformation scheme of component systems into Uppaal processes. First, we describe an ADL for the specification of the structural and the behavioral properties of component systems with aspects. Second, we describe a set of transformation rules from that ADL into Uppaal processes, we describe what a component system is and how aspects are bound and composed to the system.

6.2.1 ADL description of component systems

In this section we describe an ADL that enables the definition of both structural and behavioral properties of component systems. Our ADL enriches current ADL(s) with the information needed to detect and solve interferences among aspects. The behavior of primitive components and aspects is unified and explicitly specified. In addition, a set of weaving rules are provided to specify where and how aspects are woven and composed to the base system. Table 6.1 shows the BNF-like grammar of our ADL. In the table, *id* refers to general identifiers, *cId*, *itfId*, and *svId* refer to component, interface and service identifiers. In addition, *asId*, *pctId*, *opId*, and *locId* refer to aspect, pointcut, operator, and location identifiers, respectively. Finally, we use t to refer to data types.

		Component System Architecture Specification
Architecture	::=	**system** *id* ⟨*Interfaces*⟩ ⟨*Components*⟩ ⟨*Attachments*⟩ [⟨*Aspects*⟩ ⟨*Weavings*⟩] [⟨*Ops*⟩]
Interfaces	::=	(**interface** *id* {(@(**syn** | **asyn**) [@**rtc** "["**int**,**int**"]"] *svId*)$^+$})$^+$
Components	::=	*primitive* | *composite* | ⟨*Components*⟩; ⟨*Components*⟩
Primitive	::=	**primitive** *id* [($n :\geq 2$)] { ⟨*Template*⟩ **computation** ⟨*Behavior*⟩ }
Composite	::=	**composite** *id* [($n :\geq 2$)] { ⟨*Template*⟩ **internals** cId^+ }
Attachments	::=	**binding** (**client**=$cId_1.itfId_1$ **server**=$cId_2.itfId_2$)*
Template	::=	[**attributes** (*t id;*)$^+$] [**provides** (*itfId id;*)$^+$] [**requires** (*itfId id;*)$^+$]
		Component System Behavior Specification
Behavior	::=	**process** *id* { [(*t id*)*;] [**clock** *id**;] **state** $locId^+$; **init** *locId*; **trans** ⟨*Transitions*⟩; }
Transitions	::=	(*locId* −> *locId* {[⟨*Guard*⟩] [⟨*Sync*⟩] [⟨*Assign*⟩]})$^+$
Guard	::=	**guard** *bexp*
Sync	::=	**sync** [(**proceed** | **skip**).][*asId*.]*itfId.svId* (**!** | **?**)
Assign	::=	**assign** *exp*
		Aspects and Aspect weaving Specification
Weavings	::=	**weave** (⟨*WRule*⟩;)$^+$
WRule	::=	*asId* $(pctId, vexp)^+$ | *opId* ⟨*WRule*⟩⟨*WRule*⟩
Aspects	::=	(**aspect** *id* ⟨*Behavior*⟩;)$^+$
Ops	::=	(**operator** *id* ⟨*Behavior*⟩;)$^+$

Table 6.1: ADL description language for aspectualized component systems

According to the above ADL specification, a component system architecture (top part of Table 6.1) is defined as a set of interfaces (*Interfaces*), components (*Components*), attachments (*Attachments*), and optionally, aspects (*Aspects*), a set of weaving rules (*Weavings*) and a set of reusable composition operators (*Ops*). Each interface is defined by an identifier, and a set of service signatures, each of which is annotated with (@**syn** or @**asyn**) to indicate whether the service is synchronous or asynchronous, respectively, and optionally, a timing constraint interval (@**rtc**) indicating the lower and the upper bound time of the execution of the service. We distinguish two kinds of components: primitives (*Primitive*) and composites (*Composite*). A primitive component is defined with an identifier, a set of attributes, two sets of provided and required interfaces and a behavior indicated with the **computation** keyword. Compared with a primitive, a composite does not have a behavior, instead a set of its internal components are indicated within the **internals** keyword. Since different instances may exist in the component system configuration, an indication of the number of instances for each component is optionally indicated with a natural number ($n :\geq 2$) that follows the component name. The set of attachments defines the configuration of the system by setting down all the connections between components. Inspired from Fractal [Bruneton 2004], a connection binds a component required interface (*i.e.,* **client**) to a component provided interface (*i.e.,* **server**). The weaving part of the ADL description, indicates which

aspect should be woven to the system and how aspects are composed using binary composition operators. Both aspects and composition operators are defined with an identifier and an abstract behavior (middle part of Table 6.1). By abstract behavior aspect pointcuts are denoted with abstract names that should be replaced with concrete join points at the weaving stage (see section 6.2.6). To describe behaviors we adopt Uppaal XTA-like notation [Larsen 1997]. Accordingly, a behavior is indicated with a (**process**) keyword followed by potential declarations of local variables, clocks and a set of transitions. Each transition indicates the start and the end location, and a transition label. A transition is decorated with a guard, a synchronization channel, and a sequence of assignments. A guard is a predicate (*i.e.,* boolean expression *bexp*), its satisfaction enables the transition. For channel labels, we adopt the following notations in the ADL specification: a channel label is a concatenation of the interface and the service identifiers. In addition, a channel label can be prefixed with two predefined keywords (**proceed** and **skip**) to indicate the actions taken by an aspect. Assignments are a sequence of clock and/or variable assignments, they can also represented by calls to one or more C function. Finally, for aspect weavings (bottom part of Table 6.1), an aspect is associated with a mapping (*pctId*, *vexp*) where: *pctId* is an abstract pointcut identifier used in the aspect behavior specification, and *vexp* is a VIL expression (see chapter 5) used to define concrete join points that correspond to *pctId*. For better understanding of the ADL specification, here we describe an excerpt of the crane example. Listing 6.2 shows the specification of the `Crane` composite component.

```
1  system CraneSystem
2    interface IEngine {@sync moveLeft(Speed); @sync moveRight(Speed);}
3    interface IArm {@sync moveUp(Speed); @sync moveDown(Speed);}
4    interface IMagnet {@async setOn(); @async setOff();}
5    // other interfaces
6    primitive Engine {
7      provides IEngine iEngine;
8      requires IArm iArm;
9      // behavior
10   }
11   primitive Arm {
12     provides IArm iArm;
13     requires IMagnet iMagnet;
14     // behavior
15   }
16   composite Crane {
17     provides IEngine iEngine;
18     requires IMagnet iMagnet;
19     internals Engine, Arm;
20   }
21   binding client Engine.iArm server Arm.iArm,
22           client Crane.iEngine server Engine.iEngine,
23           client Arm.iMagnet server Crane.iMagnet;
24   // other attachments
25   aspect Performance {//Behavior}
26   weave Performance (pct, "Crane");
```

Listing 6.2: An excerpt of the ADL description of the crane example

In the above listing, three interfaces are declared with the signatures of their services (line 2-4). For example, the `IEngine` interface (line 2) defines two synchronous

services named `moveLeft` and `moveRight`, respectively. Two primitive components, the `Engine` (lines 6-10) and the `Arm` (lines 11-15), are described with their provided and required interfaces. The `Crane` composite component is described (line 16-20) with its interfaces and internals. The attachment description in (line 21) indicates that the `iArm` interface of the `Engine` component is bound to the `iArm` interface of the `Arm` component. The weaving declaration (line 26) indicates that the `Performance` aspect (line 25) is bound to the system and its abstract pointcut `pct` should be replaced by the concrete join point defined by the VIL expression `"Crane"` which indicates all the services (synchronous or asynchronous) of all the interfaces (provided and required) of the component named `Crane`.

The next sections detail how the above ADL description can be transformed into Uppaal processes. For this aim, we define a set of helper functions that introspect component architectures and provide information about the component structure: *primitives* and *composites*, two functions that introspect the whole component architecture and return the set of primitive and composite component identifiers, respectively, *services*, a function that returns the set of service identifiers defined in a given interface type identifier, *name*, a function that returns the name of any software entity (*i.e.,* component, aspect, interface, behavior) in a given architecture, *synchronous*, a function to check whether a given service name is declared synchronous in a given interface type, *attachments*, a function that returns the set of bindings defined in a given component architecture, *copy*, a function that makes a copy of a behavior of a software entity with a given new name, and *nbSync* and *nbAsync*, functions that return the number of synchronous and asynchronous services declared in a given interface description, respectively.

$$
\begin{array}{ll}
\mathit{primitives,\ composites} & : \mathit{Architecture} \longrightarrow \mathit{ID}^* \\
\mathit{services} & : \mathit{ID} \longrightarrow \mathit{Architecture} \longrightarrow \mathit{ID}^* \\
\mathit{name} & : (\mathit{Component} + \mathit{Interface} + \mathit{Behavior}) \longrightarrow \mathit{Architecture} \longrightarrow \mathit{ID} \\
\mathit{synchronous} & : \mathit{Interface} \longrightarrow \mathit{ID} \longrightarrow \mathit{Architecture} \longrightarrow \mathit{Boolean} \\
\mathit{attachments} & : \mathit{Architecture} \longrightarrow \mathit{Attachments} \\
\mathit{copy} & : \mathit{Behavior} \longrightarrow \mathit{ID} \longrightarrow \mathit{Behavior} \\
\mathit{nbSyn,\ nbAsyn} & : \mathit{ID} \longrightarrow \mathit{Architecture} \longrightarrow \mathbb{N}
\end{array}
$$

6.2.2 Formalization of primitive components

We model each primitive component as a Uppaal process. Primitive components, in the ADL, are defined with their behavior specifications, those specifications should be transformed into Uppaal-XTA form. In our ADL, the behavior specification is chosen to be a subset of the Uppaal-XTA and hence minimum adaptations are needed. In particular, each channel label in the ADL consists of the interface and the service names. In our formalization we adopt the following notation for channel labels: each label is a concatenation of the component, the interface and the service identifiers separated with "_" symbol. This notation helps on modeling component bindings (see Section 6.2.4). Accordingly, all the channel labels in the

specification should be prefixed by the component name. In addition, when more than one instance are required, the channel labels are suffixed with "`[id]`" indicating the instance reference of each component; where `id` is a constant that ranges over `[1..n]` and `n` is the indicated number of instances for the component in the specification. In Uppaal XTA, this number should be declared as a parameter of the template modeling the component. The following listing describes the general rule that generates a complete Uppaal-XTA template from the ADL specification of each primitive component declaration where `p [`y `/` x`]` denotes a substitution of each occurrence of x in `p` by y.

```
1 P : Primitive → UppaalTemplate
2 P⟦primitive cId temp computation cpt⟧= cpt[cId_itfId_svId / itfId.svId]
3 P⟦primitive cId (n) temp computation cpt⟧= let p= copy(cpt, cId(const id : [1..n]))
4                                            in p[cId_itfId_svId[id] / itfId.svId]
```

Listing 6.3: primitive component transformation rule

Listing 6.3 shows how to transform the two available forms of ADL specification of primitive components into Uppaal processes. The first form (*i.e.,* primitive with a single instance) is transformed by substituting each channel label of the form `itfId.svId`, in the behavior specification, by `cId_itfId_svId` (prefixing the original label with the component identifier cId, and replaces the default separator "." with "_"). The second form (*i.e.,* primitive with multiple instances) is transformed by making a copy of the original behavior using the `copy()` function, naming the new copy `cId(const id : [1..n])`. This enables the instantiation of the template (`n`) times. In addition, we substitute each channel label `itfId.svId` by `cId_itfId_svId[id]`. This enables to synchronize the desired component instances.

6.2.3 Formalization of composite components

A composite is modeled as a set of Uppaal processes, one for each bound interface. Each template of those processes has a central initial location and a set of directed cycles from and to that location. Each cycle describes one service. Asynchronous services are represented by cycles of two transitions: receives a message ($\texttt{cId}_1\texttt{\_itfId}_1\texttt{\_s}_i$?), then forwards it ($\texttt{cId}_2\texttt{\_itfId}_2\texttt{\_s}_i$!) (Listing 6.4 line 16-17). Synchronous services are represented by cycles with four transitions: receives a message ($\texttt{cId}_1\texttt{\_itfId}_1\texttt{\_s}_i$?), forwards it ($\texttt{cId}_2\texttt{\_itfId}_2\texttt{\_s}_i$!), waits for the reply ($\texttt{E\_cId}_2\texttt{\_itfId}_2\texttt{\_s}_i$?), and forwards the reply ($\texttt{E\_cId}_1\texttt{\_itfId}_1\texttt{\_s}_i$!) (Listing 6.4 line 11-14). When the composite has multiple instances, similar to primitives, we suffix the channel labels of the component in question with "`[id]`" (Listing 6.4 line 20-35). The following is the complete transformation rule of a composite component from the ADL specification of composites.

```
1 C : Composite → Architecture → UppaalTemplate*
2 C⟦composite cId1 temp internals cIds⟧ a=
3   ∀itfId1 ∈ interfaces(cId1),
4   ∃(client= cId1.itfId1 server= cId2.itfId2) ∈ attachments(a):
5       process cId1_itfId1() {
6         state l0,..,lk; % k =nbSync(itfId1, a)*3 + nbAsync(itfId1, a)
7         init l0;
```

```
8          trans
9          ∀ s_i ∈ services(itfId_1, a) {
10         if (synchronous(s_i, itfId_1, a)) {
11           l_0   -> l_i1 {sync cId_1 _itfId_1 _s_i ?;},
12           l_i1  -> l_i2 {sync cId_2 _itfId_2 _s_i !;},
13           l_i2  -> l_i3 {sync E_cId_2 _itfId_2 _s_i ?;},
14           l_i3  -> l_0  {sync E_cId_1 _itfId_1 _s_i !;};
15         } else {
16           l_0   -> l_i1 {sync cId_1 _itfId_1 _s_i ?;},
17           l_i1  -> l_0 {sync cId_2 _itfId_2 _s_i !;};
18         }
19       };
20 C⟦composite cId_1 (n) temp internals cIds⟧ a=
21   ∀(client= cId_1.itfId_1 server= cId_2.itfId_2) ∈ attachments(a)
22       process cId_1 _itfId_1(const id : [1..n]) {
23         state l_0,..,l_k ; % k =nbSync(itfId_1, a)*3 + nbAsync(itfId_1, a)
24         init l_0 ;
25         trans
26         ∀ s_i ∈ services(itfId_1) {
27         if (synchronous(s_i, itfId_1)) {
28           l_0   -> l_i1 {sync cId_1[id] _itfId_1 _s_i ?;},
29           l_i1  -> l_i2 {sync cId_2 _itfId_2 _s_i !;};
30           l_i2  -> l_i3 {sync E_cId_2 _itfId_2 _s_i ?;};
31           l_i3  -> l_0  {sync E_cId_1[id] _itfId_1 _s_i !;};
32         } else {
33           l_0   -> l_i1 {sync cId_1[id] _itfId_1 _s_i ?;},
34           l_i1  -> l_0 {sync cId_2 _itfId_2 _s_i !;};
35         }
36       };
```

Listing 6.4: composite component generation rule

For example, let us consider the `Crane` composite component. This component has two interfaces: `iEngine` and `iMagnet` (see listing 6.2 line 17-18). The interface `iEngine` has two synchronous services: `moveLeft` and `moveRight` (Listing 6.2 line 2) for moving the engine left and right respectively. Figure 6.2 depicts the generated Uppaal template for `iEngine` interface of the `Crane` component (two cycles of four transitions each).

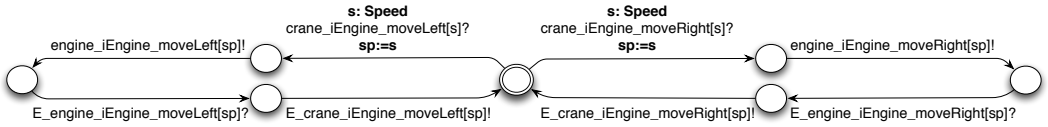

Figure 6.2: The generated Uppaal template for the `iEngine` interface of the `Crane` component

The `iMagnet` interface has two asynchronous services: `setOn` and `setOff` (Listing 6.2 line 4) for activating and deactivating the magnet field respectively. Figure 6.3 depicts the generated Uppaal template modeling the `iMagnet` of the `Crane` component (two cycles of two transitions each).

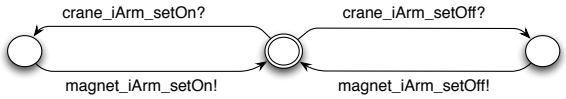

Figure 6.3: The generated Uppaal template for the `iMagnet` interface of the `Crane` component

6.2.4 Formalization of component bindings

Component bindings can be modeled either as separate Uppaal processes that receive channels from required interfaces and forward them to their bound provided interfaces, or by renaming. In our approach we adopt the second solution for minimum state number generation. By renaming, a bound interface $\texttt{itfId}_1$ of a component $\texttt{cId}_1$ to an interface $\texttt{itfId}_2$ of a component $\texttt{cId}_2$ is modeled by replacing each channel label occurrence $\texttt{cId}_1\texttt{\_itfId}_1\texttt{\_s}$ in the template of $\texttt{cId}_1$ by $\texttt{cId}_2\texttt{\_itfId}_2\texttt{\_s}$, for each service name `s`. This synchronizes the channels of two bound components. Listing 6.5 describes the general binding rule.

```
1 ℬ : UppaalTemplate → Architecture → UppaalTemplate
2 ℬ⟦p⟧ a= let cId1= name(p)
3          in ∀ (client= cId1.itfId1 server= cId2.itfId2) ∈ attachments(a),
4             ∀s ∈ services(itfId1,a) p[cId2_itfId2_s / cId1_itfId1_s];
```

Listing 6.5: binding function

6.2.5 Component systems

A complete component system without aspects is modeled as the parallel composition of all the processes modeling the components of the architecture. The primitive components are adapted to follow the Uppaal-XTA form and bound to each other using the binding function, while composite templates are automatically generated from the ADL specification. Formally, the component system of an architecture a is:

$$\mathcal{S}[\![a]\!] = \|_{\forall c \in primitives(a)}\ \mathcal{B}[\![\mathcal{P}[\![c]\!]]\!]a\ \|_{\forall t \in attachments(a)}\ \mathcal{C}[\![t]\!]a$$

6.2.6 Aspect weaving

The behavior of aspects is already described in the ADL specification following Uppaal-XTA form. The aspect behavior defines a set of cycles from and to the initial location, each of which describes the behavior of an aspect for an abstract pointcut `pctId`. The proceed and the skip actions taken by an aspect for each join point `jp` are explicitly modeled by (`proceed_jp`) and (`skip_jp`) channel labels, respectively. However, the behavior is abstract and should be instantiated for concrete join points. In our model, pointcuts are defined using VIL in a declarative style. VIL interprets and transforms the pointcut expressions into tuples of the

form *(cId, itfId, svId)* (*i.e.,* a component, an interface and a service identifiers). In addition, a mapping *(pctId,exp)* from an abstract pointcut to an expression describing the concrete join points is given in the ADL specification for each aspect (see the weaving clause in Table 6.1). In the instantiation process, for each mapping *(pctId,exp)*, we use VIL to interpret the expression *exp* and returns a set of tuples of the form *(cId, itfId, svId)*. For each tuple, we make a copy of the cycle denoting *pctId* in the aspect abstract behavior. Then, we replace each *pctId* occurrence by `cId_itfId_svId` from the tuple. Listing 6.6 describes this instantiation process of Uppaal templates from the ADL specification of aspect behaviors. In the listing, the `duplicateTransitions(asId,pctId)` function, as its name indicates, duplicates the set of transitions forming a cycle in the behavior specification *asId* where *pctId* appears as a channel label of at least one transition, and $\mathcal{V}$ denotes VIL evaluation function of pointcut expressions. For composed aspects, we instantiate each aspect individually using its defined mapping and we instantiate the composition operator. The composition operator is duplicating the `lhs` cycle for each join point defined for the left-hand side aspect only, the `rhs` for each join point of the right-hand side aspect only and `lrhs` for each shared join point. Finally, we replace each occurrence of `left` and `right` in the behavior specification of the operator by the name of left and right-hand side aspect identifiers, respectively.

```
1  ℐ : Weavings → Architecture → UppaalTemplate*
2  ℐ⟦weave asId map⟧ a= {
3    process asId {
4      // the declaration given in bevavior(asId)
5      trans
6        ∀(pctId, e) ∈ map :
7        let jps = 𝒱⟦e⟧(a)
8        in ∀(cId, itfId, svId) ∈ jps :
9        duplicateTransitions(asId, pctId)[cId_itfId_svId / pctId]
10   }
11 }
12 ℐ⟦weave w1;w2⟧ a= ℐ⟦weave w1⟧ a ∪ ℐ⟦weave w2⟧ a
13 ℐ⟦weave opId w1 w2⟧ a=
14   let p= ℐ⟦weave opId {(lhs,⊕e_w1),(rhs,⊕e_w2),(lrhs,(⊕e_w1)⊗(⊕e_w2))}⟧ a
15   in {ℐ⟦w1⟧ a, ℐ⟦w2⟧ a, p[name(w1) / left; name(w2) / right]}
```

Listing 6.6: Instantiation rule

In order to synchronize component processes with the one modelling an aspect, a set of locations and transitions have to be added to component specifications. This extension ensures that each intercepted service call is forwarded to the aspect process, which executes its behavior and returns either `proceed` or `skip`. In the former case, the extension ensures that the service call reaches its target and continues its original path. In the latter case, the extension ensures that the service call is skipped by returning to the initial location if the service is asynchronous. If the service is synchronous, all the actions between the begin and the end events of the service call are ignored. The following listing shows how the component processes are adapted to forward the join points to the aspect process (line 9,22), to wait for the aspect action (lines 9-11,23-24) and to behave accordingly either by continuing the original path (line 10,23) or returning to the location that ignores the execution of the join

point (lines 11-12,24). For the case of composed aspects, the component templates are adapted so that all the join points are redirected to the operator rather than to individual aspects (line 30-31).

```
1  R : Weavings → Architecture → UppaalTemplate*
2  R⟦weave asId map⟧ a= S(a)[
3    ∀(pct, e) ∈ map, ∀(cId, itfId, svId) ∈ V⟦e⟧:
4      let p = process(cId, a)
5      in
6        if synchronous(itfId, svId, a)
7          p[(
8            l_i -> l_i1 : {X_1 sync cId_itfId_sId? X_2},
9            l_i1 -> l_i2 : {sync asId_itfId_svId!},
10           l_i2 -> l_j : {sync proceed_asId_itfId_svId?},
11           l_i2 -> l_k : {sync skip_asId_itfId_svId?},
12           l_k -> l_l : {Y_1 sync E_cId_itfId_svId! Y_1},
13           )/
14           (
15           l_i -> l_j : {X_1 sync cId_itfId_svId? X_2},
16           l_k -> l_l : {Y_1 sync E_cId_itfId_s! Y_2},
17           )
18         ]
19       else
20         p[(
21           l_i -> l_i1 : {X_1 sync cId_itfId_svId? X_2},
22           l_i1 -> l_i2 : {sync asId_itfId_svId!},
23           l_i2 -> l_j : {sync proceed_asId_itfId_svId?},
24           l_j -> l_0 : {sync skip_asId_itfId_svId?},
25           )/
26           (
27           l_i -> l_j : {X_1 sync cId_itfId_svId? X_2},
28           )
29         ]
30 ]
31 R⟦weave w_1;w_2⟧ a= R⟦weave w_1⟧ a ∪ R⟦weave w_2⟧ a
32 R⟦weave opId w_1 w_2⟧ a= R⟦weave opId {(lhs, ⊕e_w1), (rhs, ⊕e_w2), (lrhs, (⊕e_w1) ⊗ (⊕e_w2))}⟧ a
```

Listing 6.7: Component adaptation for aspect weaving

Aspect weaving is made in two steps: aspect instantiation and component adaptation. Thus, each aspect template is instantiated for the defined join points, and the components affected by the aspect are adapted following the above rule.

```
1 W : Weavings → Architecture → UppaalTemplate*
2 W⟦w⟧ a= I⟦w⟧ a ∪ R⟦w⟧ a
```

Listing 6.8: weaving an aspect to a component

6.3 Interference detection and resolution

Now, let us show how the Uppaal model checker is used for the detection and the resolution of aspect interferences. First, we give a set of helpful formal definitions to describe component systems and aspects. We start by defining component systems and their well-definedness, aspect correctness w.r.t a component system, and an interference between two aspects. Finally, we define whether a composition operator solves an interference.

6.3.1 Well-definedness of component systems

For interference detection, a component system comes with a set of properties that the system satisfies during its execution. Here is the formal definition we give to a component system:

Definition 1 *We define a component system* Γ *as a couple* $(a_\Gamma, \mathcal{P}_\Gamma)$ *where:*

- a_Γ*: is the ADL description of the component architecture of the system.*
- $\mathcal{P}_\Gamma$*: is a set of CTL formulae describing the properties of the behavior of the system.*

Definition 2 *A component system* $\Gamma = (a_\Gamma, \mathcal{P}_\Gamma)$ *is well defined if the parallel composition of all the processes modelling the components of the system (see §6.2.5) satisfies all the desired properties of such system:*

$$\mathcal{D}(\Gamma) \stackrel{\text{def}}{=} \mathcal{S}[\![a_\Gamma]\!] \models \mathcal{P}_\Gamma$$

Properties for the Crane Base System ($\mathcal{P}_{crane}$)	
Safe 1	`A[] not deadlock`
Safe 2	`A[] Controller.configured imply User.speed == Controller.speed`
Safe 3	`A[] Engine.left || Engine.right imply Engine.speed == Arm.speed`

Table 6.2: Properties of the crane system

Table 7.1 describes the set of properties of the crane system before its behavior is altered by aspects. The crane system is designed to be a deadlock free (*Safe 1*), to ensure that the speed recommended by the user is always considered by the controller (*Safe 2*), and both the engine and the arm moves with the same speed (*Safe 3*). When the crane system is modeled as the process shown in section 6.2, the above properties are satisfied which indicates that the crane system is well defined.

6.3.2 Correctness of aspects w.r.t component systems

The intent of each aspect should also be given as a set of CTL formulae. The intent describes the set of properties the aspect ensures when it is woven to the system. The satisfaction of these properties when the aspect is bound determines the applicability of the aspect to the base system.

Definition 3 *Given a component system* $\Gamma = (a_\Gamma, \mathcal{P}_\Gamma)$*, an aspect* $\Lambda = (aId_\Lambda, map_\Lambda, \mathcal{P}_\Lambda)$*, where* $\mathcal{P}_\Lambda$ *is the set of aspect intrinsic properties (intent),* map_Λ *is the mapping between the abstract pointcuts of the aspect behavior and concrete ones described in VIL, and* $\mathcal{P}_\Gamma^\Lambda \subseteq \mathcal{P}_\Gamma$ *is the set of the system properties that must be preserved after weaving* Λ*. An aspect is said to be correct with respect to a component system* Γ *if the following condition holds:*

$$\mathcal{W}[\![\textbf{weave } asId_\Lambda \; map_\Lambda]\!] \; a_\Gamma \models \mathcal{P}_\Gamma^\Lambda \wedge \mathcal{P}_\Lambda$$

If an aspect is not correct w.r.t a component system, it should not be applied to the system since it violates the intrinsic properties of the system ($\mathcal{P}_{\Gamma}^{\Lambda}$) or it does not ensures its intent.

Properties for the save energy aspect ($\mathcal{P}_{saveEnergy}$)	
Safe 11	`A[] (Arm.movingDown1 || Arm.movingDown2 || Arm.movingUp) && SavingEnergy.nbCarriedContainers>15 imply Arm.speed==slow`
Properties for the truck safety aspect ($\mathcal{P}_{truckSafety}$)	
Safe 21	`A[] Arm.movingDown2 && TruckSafety.loading imply Arm.s==slow`
Properties for the crane safety aspect ($\mathcal{P}_{craneSafety}$)	
Safe 31	`A[] Arm.temp>60 || Engine.temp>60 imply Crane.stopped`

Table 6.3: The intent of the composable aspects

Table 6.3 describes the intents of three aspects to be woven to the crane system. The `SaveEnergy` aspect ensures that when the arm is moving up or down and the number of carried containers reaches a threshold number (15 in this case), the arm moves slow (*Safe 11*). The `TruckSafety` aspect ensures that when the arm is moving down for the second time when it is loading a container, the arm moves slow (*Safe 21*). Finally, the `CraneSafety` aspect ensures that when the temperature of the engine or the arm exceeds 60 degree, the crane is stopped (*Safe 31*). When the `SaveEnergy` and the `TruckSafety` aspects are woven to the crane system, the $\mathcal{P}_{crane}^{SaveEnergy}$ and $\mathcal{P}_{crane}^{TruckSafety}$ are defined to be all the properties except *Safe 2* since these aspects change the speed of the arm component independently of the engine. While, in the case of the `CraneSafety` the $\mathcal{P}_{crane}^{CraneSafety}$ is simply ($\mathcal{P}_{crane}$). The individual weaving of these aspects shows that they are all individually correct with respect to the crane system. In the sense that they validate both the associated system intrinsic properties and their own intents.

6.3.3 Interference and Interference-freedom of aspects

Extending components with several aspects may give rise to interferences. Two aspects are interference-free with respect to a base program, if: (1) each aspect is correct with respect to the base program, (2) when both aspects are bound to the system, the result process satisfies all the properties of the underlying aspects and the system intrinsic properties defined for both aspects. Formally:

Definition 4 *Given a base system* $\Gamma = (a_{\Gamma}, \mathcal{P}_{\Gamma})$*, two aspects* $\Lambda_1 = (asId_{\Lambda_1}, map_{\Lambda_1}, \mathcal{P}_{\Lambda_1})$*,* $\Lambda_2 = (asId_{\Lambda_2}, map_{\Lambda_2}, \mathcal{P}_{\Lambda_2})$*.* Λ_1 *and* Λ_2 *are interference-free if the following conditions hold:*

1. *the base system is well defined:* $\mathcal{D}(\Gamma)$
2. *the composition is correct w.r.t* Γ*:*
   (a) $map_{\Lambda_1} \cap map_{\Lambda_2} = \phi$*:*
   $\mathcal{W}[\![\textbf{weave}\ (asId_{\Lambda_1}\ map_{\Lambda_1}); (asId_{\Lambda_2}\ map_{\Lambda_2})]\!]\ a_{\Gamma} \models \mathcal{P}_{\Gamma}^{\Lambda_1} \wedge \mathcal{P}_{\Gamma}^{\Lambda_2} \wedge \mathcal{P}_{\Lambda_1} \wedge \mathcal{P}_{\Lambda_2}$

(b) $map_{\Lambda_1} \cap map_{\Lambda_2} \neq \phi$:
$\mathcal{W}[\![\textbf{weave}\ Seq\ (asId_{\Lambda_1}\ map_{\Lambda_1})\ (asId_{\Lambda_2}\ map_{\Lambda_2})]\!]\ a_\Gamma \models \mathcal{P}_\Gamma^{\Lambda_1} \wedge \mathcal{P}_\Gamma^{\Lambda_2} \wedge \mathcal{P}_{\Lambda_1} \wedge \mathcal{P}_{\Lambda_2}$

Where $\mathcal{P}_\Gamma^{\Lambda_1}$ and $\mathcal{P}_\Gamma^{\Lambda_2}$ denote the set of intrinsic properties of the system defined for Λ_1 and Λ_2, respectively.

In our crane example, when the `saveEnergy` and the `truckSafety` are woven to the crane, all their properties and the associated intrinsic properties defined for the crane are satisfied which indicates that they are interference-free. However, when the `saveEnergy` and the `craneSafety` are woven, *Safe 11* is violated which indicates that these two aspects are interfering.

6.3.4 Composition operators solving Interferences

A composition operator solves an interference between two aspects if the instantiation for these two aspects and the composition of its template to the system, the interference disappears. Formally:

Definition 5 *Given a base system* $\Gamma = (a_\Gamma, \mathcal{P}_\Gamma)$ *and two interfering aspects* Λ_1 *and* Λ_2 *such that:* $\Lambda_1 = (asId_{\Lambda_1}, map_{\Lambda_1}, \mathcal{P}_{\Lambda_1})$, *and* $\Lambda_2 = (asId_{\Lambda_2}, map_{\Lambda_2}, \mathcal{P}_{\Lambda_2})$. *A composition operator opId solves an interferences if the following condition hold:*

$$\mathcal{W}[\![\textbf{weave}\ opId\ (asId_{\Lambda_1}\ map_{\Lambda_1})\ (asId_{\Lambda_2}\ map_{\Lambda_2})]\!]\ a_\Gamma \models \mathcal{P}_\Gamma^{\Lambda_1} \wedge \mathcal{P}_\Gamma^{\Lambda_2} \wedge \mathcal{P}_{\Lambda_1} \wedge \mathcal{P}_{\Lambda_2}$$

We should mention here, that in our proposal, when two aspects share common join points (*i.e.,* service call events), they are composed sequentially by instantiating the `Seq` template. When the default sequential composition does not solve the problem, a diagnosis trace is reported to the user to find out the source of the error and potentially choose another operator from our list or define a new one in a similar way. For the crane example, the `truckSafety`, the `saveEnergy` and the `craneSafety` aspects share the `load` and `unload` service call events. It is observed that `Seq(saveEnergy, truckSafety)`, reports no error, which indicates that the default composition of these two aspects is correct with respect to the crane system. However, `Seq(saveEnergy, craneSafety)` reports an interference between these two aspects with a diagnosis trace. Analyzing the reported trace, shows that when the `Seq` operator is used in this case, the `load` and `unload` service calls are forwarded to the `saveEnergy` after being skipped by the `craneSafety`. This causes a false value on the `nbCarriedContainer` variable maintained by the `saveEnergy` aspect. To solve this problem, another operator should be used. The required behavior of this operator is to skip service calls if the first aspect decides to skip it. This operator is named `And` operator. The use of this operator (*i.e.,* `And(craneSafety, saveEnergy)`) reports no error and thus it solves this interference.

6.4 Composition operators catalog

In this section we provide a set of abstract composition operators modeled as Uppaal templates. Those templates can be instantiated for aspects in order to solve their potential interferences. The operators are presented as patterns for aspect interference resolution. Similar to the GoF design patterns [Gamma 1995], each operator pattern is given a significant name, a motivation example, applicability cases, a structure and a semantic. The structure is given as a Uppaal template, while the semantic is presented as a table showing how the operator behaves according to the actions taken by aspects. In the semantics table (-) denotes that an aspect is not called for a given join point.

6.4.1 Fst composition pattern

Applicability: The `Fst` operator can be used in the following cases:

1. two aspects are mutually exclusive and only one aspect should be applied;
2. one aspect satisfies all the properties of another and hence only the first one should be applied;
3. an aspect behavior have to be hidden on a subset of its join points.

Motivation example: The `criticalLifting` and the `performance` aspects of the crane example are mutually exclusive: the former aspect forces the `engine` and the `arm` to move slowly while the latter forces them to move rapidly.

Structure: the `Fst` operator directly proceeds all the calls intercepted by the second aspect and forwards all the other calls to the first aspect only. Thus, only the first aspect is executed when a shared join point is intercepted and the second aspect is never called. Figure 6.4 depicts the template modelling the `Fst` operator. In the figure, the intercepted call by either the first aspect only `lhs?` or by both aspects `lrhs?` are forwarded to the first aspect only `left_lhs!` and `left_lrhs!` and the operator waits for the decision of the aspect and forwards it. The intercepted call `rhs?` denotes an intercepted service call by the second aspect only. This latter is directly proceeded (`proceed_rhs!`).

Semantic: the following table shows how the `Fst` act following the different join points (*i.e.,* shared or not) and the actions taken by the underlying aspects. In the table, (-) is used to indicate that the join point is not forwarded to the aspect in question.

6.4.2 Seq composition pattern

Applicability: the `Seq` operator is applicable in the following cases:

1. the default composition of two aspects sharing at least one join point.
2. precedence relationship between aspects.

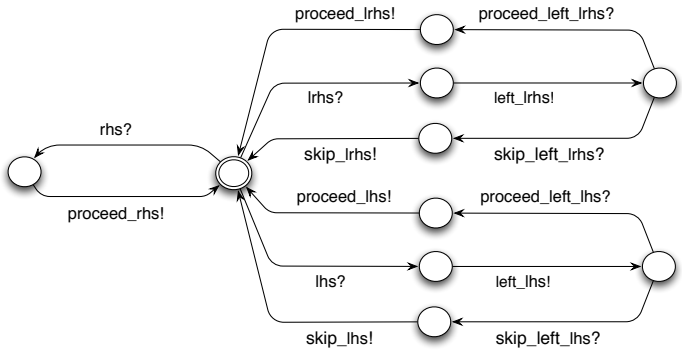

Figure 6.4: The `Fst` template

LHS	RHS	Fst(LHS,RHS)
Shared join points / LHS join points		
proceed	-	proceed
skip	-	skip
RHS join points		
-	-	proceed

Table 6.5: The semantic of the `Fst` operator

3. one aspect satisfies a subset of the properties of another and hence both aspects should be executed in a specific order.

Motivation example: The `saveEnergy` and the `truckSafety` aspects of the crane example are one-way inclusive. This is because the `truckSafety` forces the `arm` to move slowly when it is carrying a container to the truck. This is a particular case of the `saveEnergy` aspect that forces the arm to move slowly in all the cases after carrying a threshold number of containers. The `Fst` operator cannot be used in this case, because, before reaching the threshold number of carried containers, the `truckSafety` should be executed.

Structure: the `Seq` operator forwards the intercepted calls common to both aspects to the first aspect, it waits for its decision to send it to the second aspect. The call in this case is proceeded when at least one of the aspects decides to proceed it, otherwise, the service call is skipped. In addition, the intercepted calls by either aspects are forwarded to their corresponding aspects only. Figure 6.5 depicts the template modelling the `Seq` operator. In the figure, the intercepted call `lrhs?` intercepted by both aspects is forwarded to the first aspect (`left_lrhs?`), the decision of this aspect is saved in a local variable `fstAct`, then the call is forwarded to the second aspect (`right_lrhs?`). The call is proceeded when it is proceeded by the second aspect (`proceed_right_lrhs?`) or it is skipped by it and proceeded by the first aspect. Two other cycles describe forwarding non common service calls to their corresponding aspects (cycles for `lhs?` and `rhs?`).

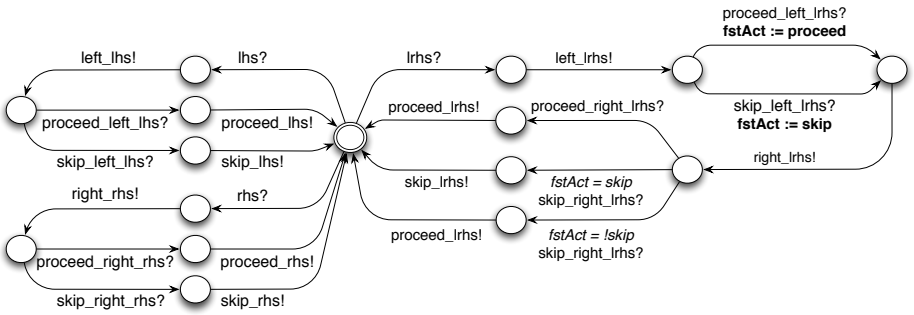

Figure 6.5: The `Seq` template

Semantic:

LHS	RHS	Seq(LHS,RHS)
Shared join points		
proceed	proceed	proceed
skip	proceed	proceed
proceed	skip	proceed
skip	skip	skip
LHS join points		
proceed	-	proceed
skip	-	skip
RHS join points		
-	proceed	proceed
-	skip	skip

Table 6.6: The semantic of the `Seq` operator

6.4.3 Cond composition pattern

Applicability: the `Cond` operator is applicable in the following cases:

1. the execution of the first aspect should always be executed while the execution of one aspect relies on state variables or any other effects generated from the execution of the other aspect (original form).
2. the execution of both aspects is based on state variables and any other predicate (variant).

Motivation example: see the next chapter Section 7.4.2.

Structure: the `Cond` operator (see Figure 6.6) forwards each intercepted call to the first aspect and maintains a predicate when the action of the aspect is received. According to the predicate, the call is forwarded to the second aspect or the action of the first aspect is directly taken. The structure in figure 6.6 describes the case of two aspects sharing join points. Another variant of this operator

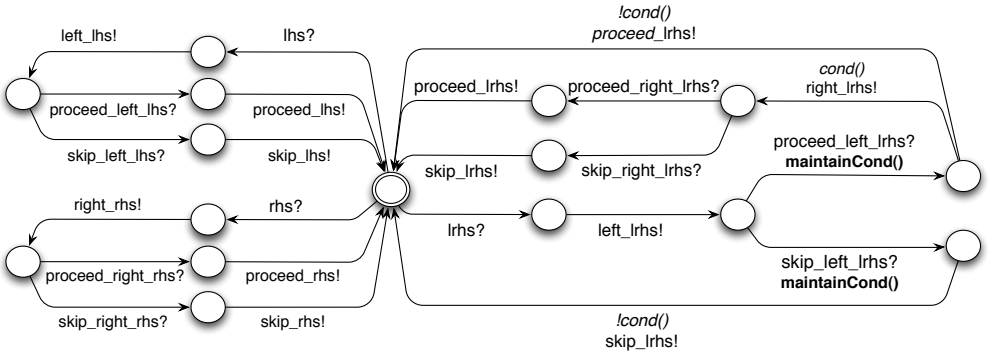

Figure 6.6: The `Cond` template

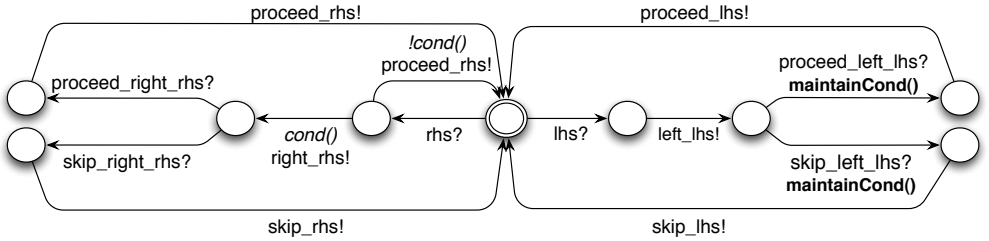

Figure 6.7: The `Cond` template (variant for non shared join points)

is designed to consider two aspects with no shared join points, it is shown in Figure 6.7.

Semantic:

Pre	LHS	RHS	Cond(p,LHS,RHS)	Post
Shared join points				
p=true	proceed	proceed	proceed	maintain(p)
	proceed	skip	skip	maintain(p)
	skip	-	skip	maintain(p)
p=false	proceed	-	proceed	maintain(p)
	skip	-	skip	maintain(p)
LHS join points				
	proceed	-	proceed	
	skip	-	skip	
RHS join points				
	-	proceed	proceed	
	-	skip	skip	

Table 6.7: The semantic of the `Cond` operator

Pre	LHS	RHS	Cond(p,LHS,RHS)	Post
LHS join points				
	proceed skip	- -	proceed skip	maintain(p) maintain(p)
RHS join points				
p=true	- -	proceed skip	proceed skip	
p=false	-	-	proceed	

Table 6.8: The semantic of the `Cond` operator (variant)

6.4.4 And composition pattern

Applicability: the `And` operator is applicable in the following cases:

1. two aspects complement each other and should only be executed in a specific order otherwise a conflict appears.
2. there is a precedence relation between two aspects and the execution of one aspect should not be in a conflict with the action taken by another.

Motivation example: It is the case for the `saveEnergy` and the `craneSafety` aspects of the crane example. The `craneSafety` may decide to skip or proceed loading an unloading containers, this should be executed first before the action is counted by the `saveEnergy` aspect.

Structure: the `And` operator is used to proceed calls only when it is proceeded by both aspects and skips it when one of the aspects skips it. If the first aspect decides to skip the call, the call is directly skipped without being forwarded to the second aspect. In addition, the `And` operator can be used to block aspect actions on join points that are not yet handled by the other aspect.

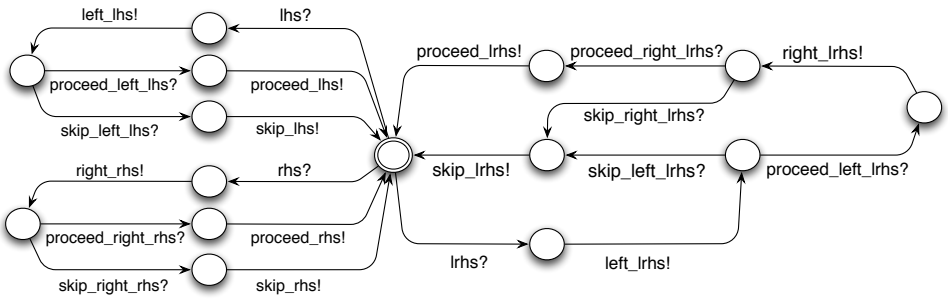

Figure 6.8: The `And` template

Semantic:

6.4.5 Alt composition pattern

Applicability: the `Alt` operator is applicable in the following case:

LHS	RHS	And(LHS,RHS)
Shared join points		
proceed	proceed	proceed
proceed	skip	skip
skip	-	skip
LHS join points		
proceed	-	proceed
skip	-	skip
RHS join points		
-	proceed	proceed
-	skip	skip

Table 6.9: The semantic of the `And` operator (variant)

- two aspects share some join points and must be executed alternately on those join points.

Motivation example: see the next chapter Section 7.4.1.

Structure: the `Alt` operator is used to alternately call aspects on shared join points. Figure 6.9 is the template showing the `Alt` operator. The shared intercepted service `lrhs?` is alternately forwarded to the first and the second aspect, respectively. The turn of each aspect for a particular join point is captured by the function `isLeft(lrhs)`. The other intercepted events `lhs` and `rhs` are forwarded to their corresponding aspects `left` and `right`, respectively.

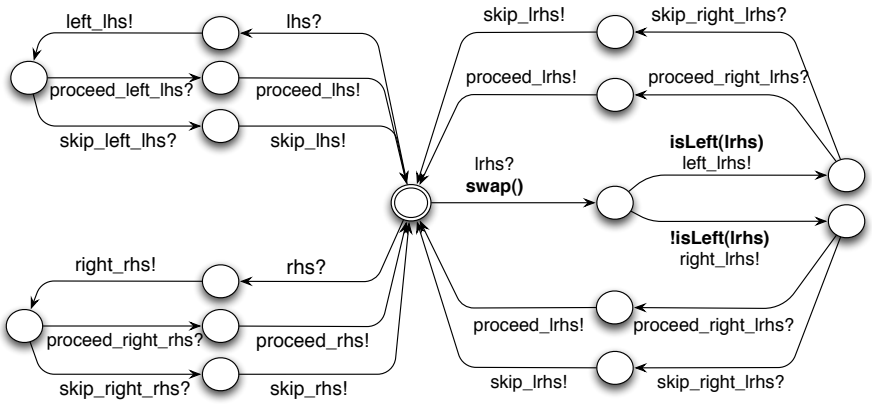

Figure 6.9: The `Alt` template

Semantic:

6.5 Conclusion

This chapter describes our contribution in the detection and the resolution of aspect interferences in component systems. The approach proposes an ADL that enriches

Pre	LHS	RHS	Alt(LHS,RHS)	Post
Shared join points				
NbOcc(thisJoinPoint)% 2=0	proceed skip	- -	proceed skip	NbOcc(thisJoinPoint)++
NbOcc(thisJoinPoint)% 2=1	- -	proceed skip	proceed skip	
LHS join points				
	proceed skip	- -	proceed skip	
RHS join points				
	- -	proceed skip	proceed skip	

Table 6.10: The semantic of the `Alt` operator

component architectures description with formal specifications of primitive component and aspect behaviors. This enables formal verification using model checkers. A set of transformation rules from the ADL to state machines are presented and exemplified with our running example. We have shown how Uppaal model checker is used to detect interferences. In addition, a set of composition operators are presented as patterns for aspect interference resolution. We have to mention that our approach does not rely neither on a specific component model nor on a particular model checker. The transformation rules are general but not exhaustive since other component models such as Sofa 2 [Bures 2006, Hnetynka 2007] may require other rules for transforming their own software architecture elements. For example, Sofa connectors are first class entities that implement different communication strategies, those connectors can be modeled as Uppaal processes implementing the desired strategy. In addition, the Uppaal model checker is used for its suitable provided features such as local variables declaration, parameter passing and process instantiation. Moreover, the set of proposed composition operators is an extendible set where other operators can be defined in a similar way. The proposed set of operators are sufficient to solve the interferences among the aspects proposed for our running example, and our case study (see Chapter 7). Finally, we should mention that the composition operators are only proposed as a support for the users since several other cases require ad hoc solutions. For example, adapt the runtime behavior of one aspect according to the presence or the absence of other aspects [Kienzle 2009].

Chapter 7

Case Study: Airport Internet Access

Contents

In this chapter, we apply our proposal to a significant and a concrete example, we show how views are defined using VIL (see **Chapter 3**), how aspects are added to the views and how interferences are detected and solved following our formal model (see **Chapter 4**). The example is developed in Fractal component model and our Fractal weaver is used to create the required views. Compared with the running example, the case study requires more properties of component models such

as multiple instances of components and dynamic reconfiguration of the system. In addition, the proposed aspects require more complicated views and more operators to solve the interferences among them.

7.1 Base System Architecture

Our case study models an airport service for providing a wireless Internet connection for passengers [Šery 2007, Adamek 2007]. Free Internet access is granted to passengers owning valid flight tickets. A passenger uses his/her flight ticket number to login and access the network for an associated time to the ticket. Three kind of users are distinguished: (1) users owning valid flight tickets who use their tickets number to access the network. (2) users owning frequent flight tickets who use their frequent flight ticket number to access the network with a permitted access time of the maximum provided time by the flight tickets of the user, and (3) regular users that do not have valid flight tickets or frequent flight tickets or have already consumed all the time provided by their tickets. They may create a temporary account and buy access time. Figure 7.1 depicts the component architecture of this application.

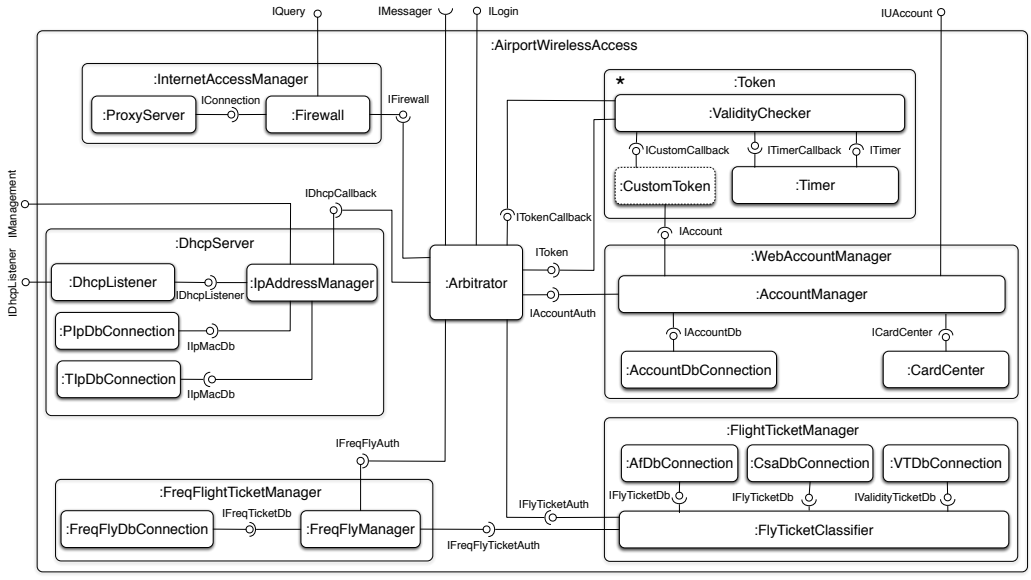

Figure 7.1: Architecture of the airport wireless access system

To connect to the airport wireless system, a user requests an IP address from the `DhcpServer` (use `IDhcpListener` interface), then it asks to login from the `AirportServer` (use `ILogin` interface). Once connected, it sends queries to the `InternetAccessManager` (use `IQuery` interface). The `InternetAccessManager` forwards users' requests to the `Firewall` that blocks unauthorized Internet connections. The requests of users with enabled IP addresses are actually sent to the Net `Proxy`.

The IP address requests are first captured by the `DhcpListener` component that delegates them to the `IpAddressManager` which provides a dynamic allocation of IP addresses. The allocated IP addresses are managed by the `TIpDbConnection` and `PIpDbConnection` components. Only one of the two above components is used for each IP address as determined by the manager using the `IManagement` interface. The `Token` component models a user session. When the `Arbitrator` receives a login request, it checks the validity of the ticket, the frequent flight ticket or the input account number, and retrieves the authorized access time from the `FlyTicketManager`, the `FreqFlyTicketManager`, or the `WebAccountManager`, respectively. Then it orders the `InternetAccessManager` to enable communications for the user, and starts a new session by instantiating a `Token` and calls the `ValidityChecker` to start the session `Timer` component. When session time elapses, the `Timer` informs the `ValidityChecker` which in turn informs the `Arbitrator`. The `Arbitrator` closes the session by calling the `InternetAccessManager` to disable the user communications, and the `DhcpServer` to disable its IP address. The `CustomToken` component is optional (*i.e.,* indicated with dashed lines) and it appears only for the sessions of regular users. When the `ValidityChecker` is informed of the end of a regular user session, it calls the `CustomToken` to update the permitted connection time of the user in the `WebAccountManager`. Regular users may create temporary accounts and pay for an indicated access time using `IUAccount` interface of the `AirportServer` component. The payment is managed by the `CardCenter` component, while the consumed time is managed by the `AccountDbConnection` component.

7.2 Aspects on Views

We want to alter the functionality of the system by introducing a set aspects. Each of which defines a new functionality that requires the adaptation of a set of scattered components over the system architecture. However, these adaptations cannot be added to the components behavior directly when their source code is not provided. The desired aspects are:

Bonus: offer free Internet access promotions to clients according to their categories.

Alert: warn clients five minutes before the end of their current sessions.

Safety: terminate sessions when their corresponding flights are taking off.

NetOverloading: block P2P access when the number of active sessions reaches 1000.

LimitedAccess: consider different access privileges according to clients age.

In the following, we detail each functionality, its required view, and the implementation of each aspect as a composable controller in Fractal component model (see **Chapter 3**).

7.2.1 The `Bonus` Aspect

Let us suppose that the airport manager decides to offer a bonus time to customers according to their types (*e.g.,* 10 minutes for valid fight ticket owners, 20 minutes for frequent flight ticket owners and only 5 minutes for regular customers). Figure 7.2 shows a scenario of how 10 minutes of bonus time are added to a valid flight ticket owner with 60 minutes as authorized connection time.

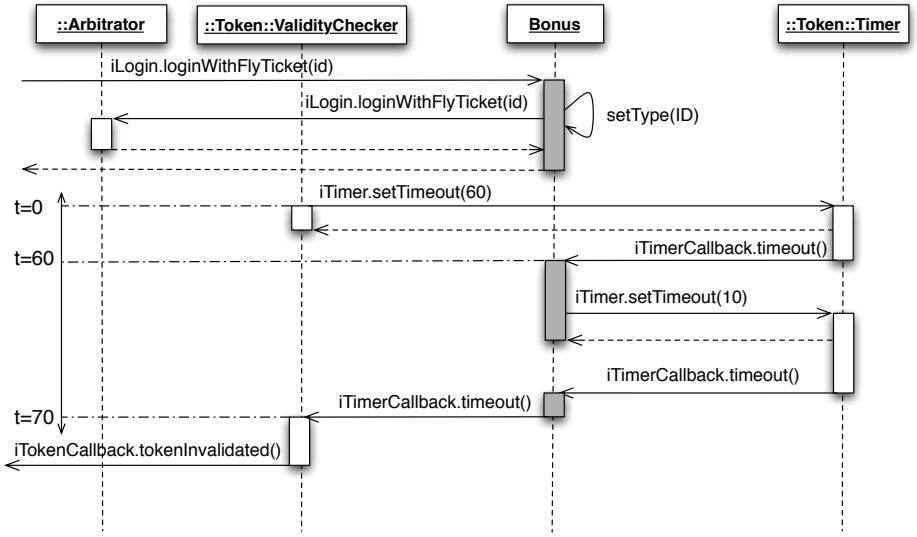

Figure 7.2: A scenario of adding a bonus to customers

The above scenario shows that the `Bonus` aspect requires the interception of the `ILogin` interface of the `Arbitrator` to determine the type of each logged in customer. In addition, it requires the interception of the `ITimerCallback` interface of `ValidityChecker` component to receive the `timeout` service call which is called by the `Timer` to inform the `ValidityChecker` of the end of the session, thus the aspect uses the `ITimer` interface to reset the `Timer` for a bonus time and proceeds the next intercepted `timeout` on the `ITimerCallback` interface. According to the above scenario, the required view must encapsulate the `Arbitrator` and the `ValidityChecker` components on the same composite. However, the required view is not fulfilled by the original architecture, and hence a transformation is needed. Following our proposed implementation for Fractal (see Chapter 3), a new composite is created and the `Arbitrator` and the `ValidityChecker`(s) components are declared as its inner components sharing them with their original parents. This view can be defined in VIL using the following expression:

$$vBonus = (\textbf{provide} \ * \ (\textbf{T} \ \{ILogin\})) \ \oplus \ (\textbf{instance} \ (\textbf{bound} \ \{Timer\}))$$

The above VIL expression states that the required view should encapsulate all the components providing the `ILogin` interface and all the instances of all the com-

ponents bound to the `Timer` component. In addition, it implicitly states that the intercepted and used interfaces are `ILogin` and all the provided and required interfaces of the component instances bound to the `Timer` (*i.e.,* `ITimer` and `ITimerCallback` in this case).

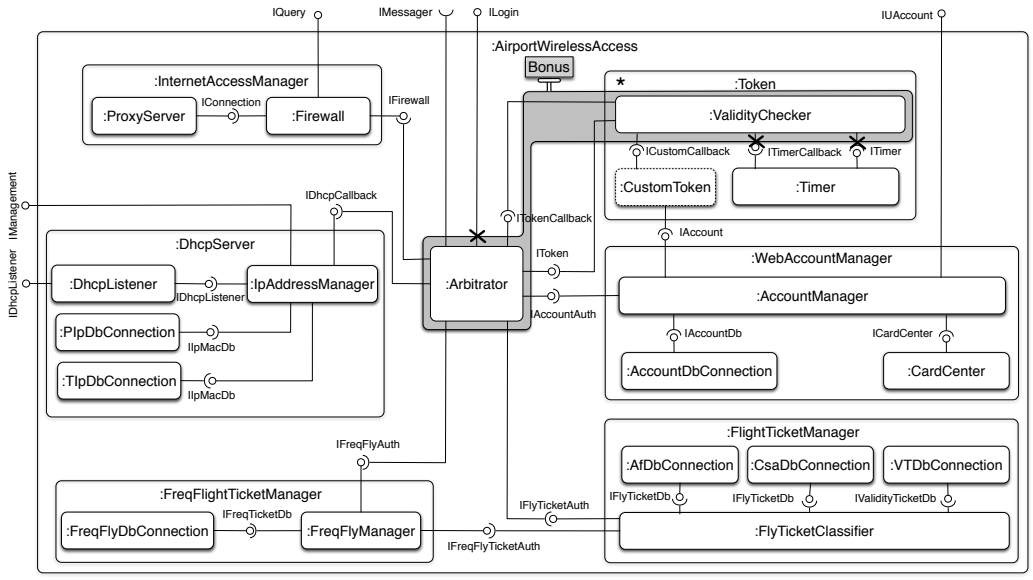

Figure 7.3: The airport system extended with `Bonus` (the `Bonus` view)

The `Bonus` aspect is modeled in Fractal as a composable controller `(Dispatcher,-Bonus)` added to the new composite (see Figure 7.3, the intercepted and used interfaces are marked with **"X"**). The `Dispatcher` controller intercepts service calls to the `ILogin` and the `ITimerCallback` interfaces of the `Arbitrator` and the `ValidityChecker` components, respectively. The `Dispatcher` intercepts a call, reifies it into a message object and sends it to the `Bonus` object by calling its `match` method. The `match` method of `Bonus` behaves as follows: when it receives a `login*` call, it stores the type of the customer (*e.g.,* `loginWithFlyTicket` implies that the customer is a flight ticket owner), when it receives the first occurrence of `timeout` (*i.e.,* the customer session should be closed), it checks the previously stored type of the customer to decide which bonus time should be affected, it resets the timer for the correspondent bonus time (by calling the `setTimeout(bonusTime)` service on the `ITimer` interface of the corresponding `Timer` component), and informs the `Dispatcher` controller that it wants to skip the call by returning a `Skip` command. That means that the `timeout`, in this case, is not actually proceeded and the session continues. If it receives a second occurrence of `timeout` from the same customer, the `match` method returns a `Proceed` command to `Dispatcher`. This causes the `Dispatcher` to call its `invoke()` method which proceeds the call and ends the current session of the user.

7.2.2 The `Alert` Aspect

Now, suppose that the airport manager decides to add a service that alerts users five minutes before their authorized time connection elapses. In this case, a session timer is initialized with five minutes less than the authorized time. When the time elapses and a timeout is generated, an alert is sent to the user and the timer is reset for five more minutes. Figure 7.4 shows a scenario of how the alert is sent to users 5 minutes before the end of their sessions.

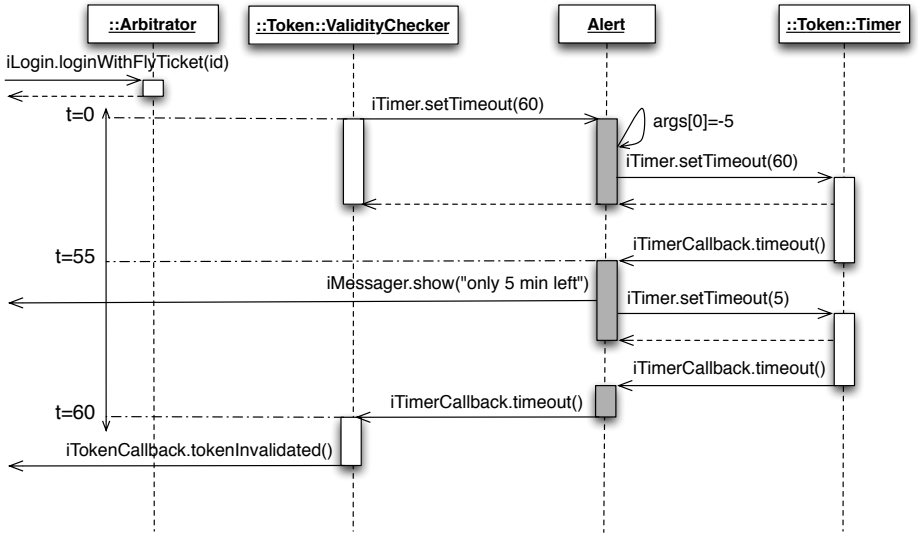

Figure 7.4: A scenario of alerting users before the end of their sessions

In order to fulfil the alert aspect requirement, the aspect requires to intercept the `ITimerCallback` interface (for the `timeout` calls) and uses the `ITimer` (to reset the timer for 5 min) and the `IMessager` interface (to send an alert message to users). Since the required interfaces belong to components scattered over the architecture (*i.e.,* the `ValidityChecker` and the `Arbitrator` components do not belong to the same composite), a view must be defined. In this case, the required view is the same as the one defined for the `Bonus` aspect except that it intercepts `IMessager` rather than `ILogin`. The following VIL expression defines the view required by the `Alert` aspect.

$$vAlert = (\textbf{require} \ * \ (\textbf{T} \ \{IMessager\})) \ \oplus \ (\textbf{instance} \ (\textbf{bound} \ \{Timer\}))$$

Similar to `Bonus`, the `Alert` aspect in Fractal is modeled as a composable controller `(Dispatcher,Alert)` added to the new composite encapsulating the `Arbitrator` and the different instances of `ValidityChecker`(s) as shown in Figure 7.5.

The `match` method of `Alert` behaves as follows: when the `setTimeout` service call is intercepted on the `ITimer` interface of the `ValidityChecker` component, the `Alert` changes the parameter value of the call (*e.g.,* 60 minutes) by subtracting a time alert (5 minutes), and proceeds the call. Thus, the `ValidityChecker` will

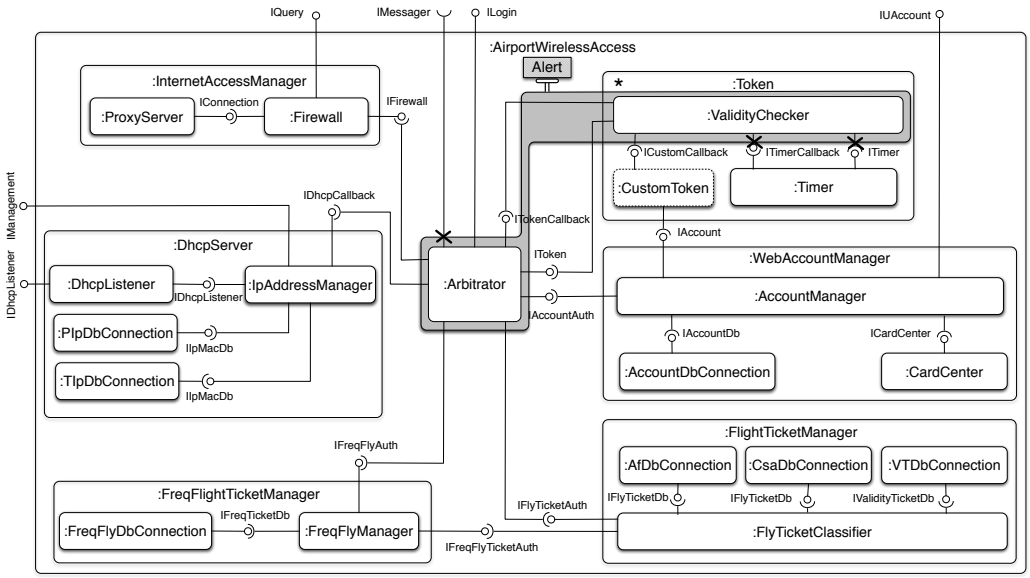

Figure 7.5: The airport system extended with `Alert`

receive a `timeout` 5 minutes before the end of the session. When the `timeout` is intercepted, the `Alert` aspect sends an alert message to the user (by calling the `show("you have only 5 min left")` on the `IMessager` interface required by the `Arbitrator`), resets the `Timer` for 5 minutes by calling `setTimeout(5)` and skips the currently intercepted `timeout`. The `Alert` proceeds the next intercepted `timeout` to end the session.

7.2.3 The `NetOverloading` Aspect

The `NetOverloading` aspect intent is to block P2P connection queries when the number of connected users exceeds a threshold number. The implementation of such aspect requires the interception of the `IQuery` interface to check whether the required IP connection is P2P and refuse the access to such queries when the number of connected users exceeds the threshold number. In order to count the number of connected users, the aspect needs to intercept the `IIpMacDb` that defines the services `add` and `remove` to save or delete the IP addresses of users when they are connected or disconnected, respectively. Thus, a new view is required for this aspect in which the `Firewall`, the `TIpDbConnection`, and the `PIpDbConnection` providing the required interfaces are encapsulated in the same component. The following is the VIL expression describing the required view:

$$vNetOverloading =$$
$$(\textbf{direct provide} * (\textbf{T} \{IQuery\})) \oplus (\textbf{provide} * \{add(IP), remove(IP)\})$$

The above expression states that the required view must encapsulate the primitive component providing `IQuery` interface (the use of **direct** keyword) and all

the components providing interfaces defining methods matching *add(IP)* and *remove(IP)*. In addition, all the methods of `IQuery` interface have to be intercepted.

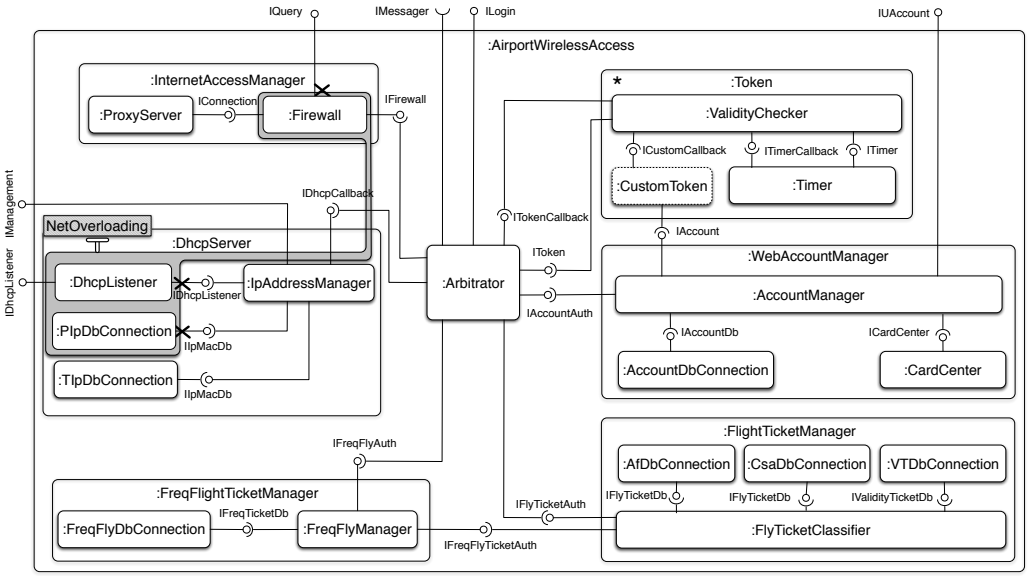

Figure 7.6: The airport system extended with `NetOverloading`

Figure 7.6 depicts the airport system after wrapping the `Firewall`, the `TIpDbConnection`, and the `PIpDbConnection` by the composable controller `(Dispatcher,NetOverloading)`. The `Dispatcher` in this case intercepts calls to the `IQuery` and `IIpMacDb`, and calls the `match` method of the `NetOverloading` ICController object which behaves as follows: each time it receives `add(IP)` (resp. `remove(IP)`) it increments (resp. decrements) the number of connected users. When it receives `connect(IP)` on the `IQuery` interface, it checks if the required IP is P2P address. If it is the case, it checks the stored number of connected users, if it reaches a predefined threshold number, it returns `Skip` to the `Dispatcher` and the call is skipped, otherwise, it returns `Proceed` and hence the connection establishes.

7.2.4 The `LimitedAccess` Aspect

The `LimitedAccess` aspect aims to prevent minors access to blacklisted websites. To fulfil this requirement, the aspect should intercept the `IQuery` interface to check whether the required IP is in the black list and block the access if the customer is minor. The age of the customer is stored in the ticket or in the account information in the corresponding databases. To access these information, the aspect should intercept the `IAccountAuth`, the `IFlyTicketAuth`, and the `IFreqFlyAuth` for each customer type, respectively. In this case, since the `Arbitrator` requires all these interfaces and requires all the information of the customer at each login request, the `Arbitrator` component is the one that should be encapsulated with the `Firewall` in

the required view (see Figure 7.7). The following is the correspondent VIL expression defining the view:

$$vLimitedAccess = (\textbf{direct provide} * (\textbf{T} \{IQuery\})) \oplus (\textbf{require} * (\textbf{T} \{*Auth\}))$$

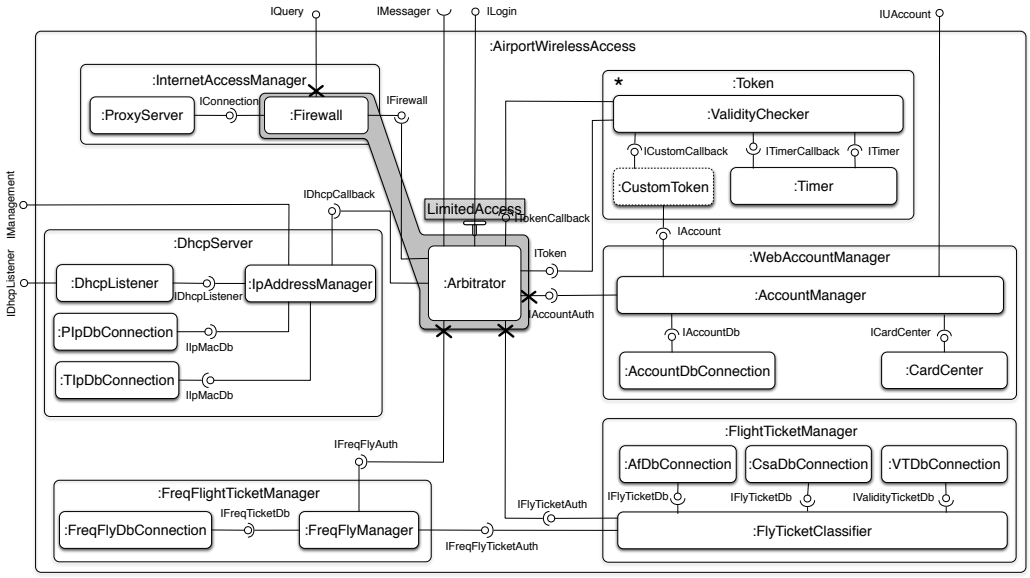

Figure 7.7: The airport system extended with `LimitedAccess`

The `match` method of the `LimitedAccess` object behaves as follows: when it receives the return value of the service call `getInfo` on one of the required interfaces by the `Arbitrator`, it stores the age of the customer with its IP address. When it receives a customer queries on the `IQuery` interface, it checks whether the requested IP connection is in the black list and the customer is minor (age <=18). If it is the case, the `match` method returns `Skip` to the `Dispatcher` to ignore the call, otherwise it returns `Proceed` and the connection establishes.

7.2.5 The `Safety` Aspect

The `Safety` aspect aims to automatically end the sessions of current connected customers at taking off of their planes. Implementing this feature, the aspect requires the `IFlyTicketDb` to access the information about the exact taking off of planes, the `IToken` interface to end the session and the `ILogin` interface to map the customer IP with the flight ticket number. Thus, the required view must ensure that the `Arbitrator` and the `FlyTicketClassifier` belong to the same composite as indicated in Figure 7.8. This view is obtained by interpreting the following VIL expression:

$$vSafety = (\mathbf{require} * \{getTicket(ID)\}) \oplus (\mathbf{provide} * (\mathbf{T}\ \{ILogin\})) \oplus (\mathbf{require} * (\mathbf{T}\ \{IToken\}))$$

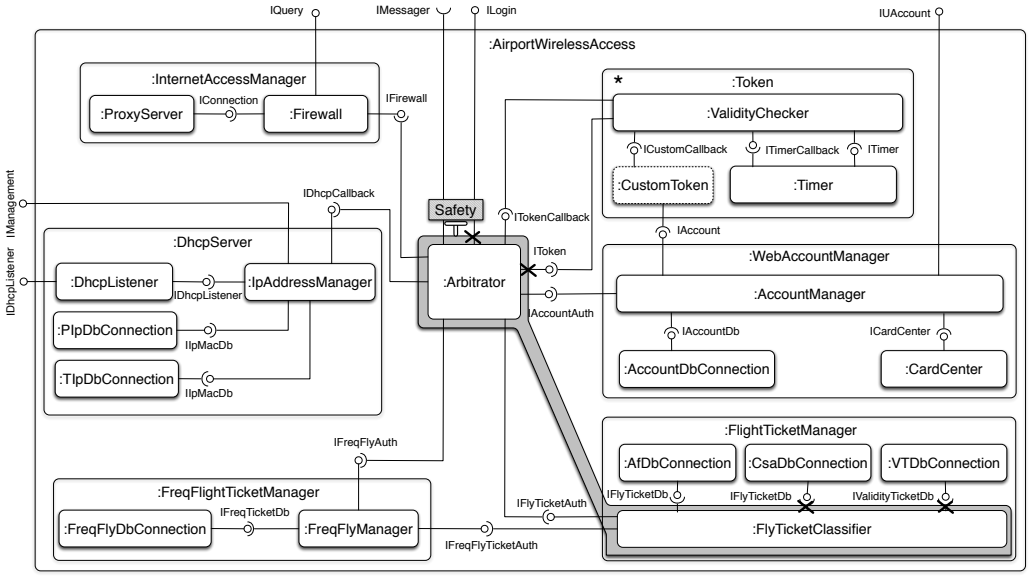

Figure 7.8: The airport system extended with `Safety`

7.3 Formal Specification in Uppaal

For formal detection of interferences among the above defined aspects, the complete system is modeled as Uppaal processes following the model presented in chapter 4: (1) a Uppaal process is generated for each primitive component from its behavior specification given in the ADL, (2) a set of Uppaal processes are generated from the ADL specification for each bound interface of a composite component, and (3) a renaming process is executed to bound components (*i.e.,* synchronize processes modelling components on the bound interfaces).

```
1 system airportInternetAccess {
2   interface ITimer {@sync setTimeout(int)}
3   interface ITimerCallback {@async timeout()}
4   interface IToken {@sync startToken()}
5   interface ITokenCallback {@async tokenInvalidated()}
6   // other interfaces
7   primitive Timer (n) {
8     provides ITimerCallback iTimerCallback;
9     requires ITimer iTimer;
10    // behavior
11  }
12  primitive ValidityChecker (n) {
13    provides ITimer iTimer, IToken iToken;
14    requires ITimerCallback iTimerCallback,
15             ITokenCallback iTokenCallback;
16    // behavior
17  }
18  composite Token (n) {
19    provides ITokenCallback iTokenCallback;
20    requires IToken iToken;
21    internals Timer(1), ValidityChecker(1);
22  }
23  binding
24    client ValidityChecker.iTimer server Timer.iTimer;
25    client ValidityChecker.iTokenCallback server Token.iTokenCalback
26    client Token.iToken server ValidityChecker.iToken
27    // other attachments
28 }
```

Listing 7.1: An excerpt of the ADL of the airport system example

Listing 7.1 shows an excerpt of the ADL specification of the airport Internet access example. In the listing, four interfaces are declared with the signatures of their services (line 2-5). For example, the `ITimerCallback` interface defines only one asynchronous service named `timeout`. Two primitive components, the `Timer` and the `ValidityChecker`, are described (lines 7-11, 12-17, respectively) with their provided and required interfaces. The `Token` composite component is described (line 18-22) with its interfaces and internals. Since several tokens can be created, the component is parametrized with n (the maximum number of instances). In the internals part declaration, the `Timer` and the `ValidityChecker` are parametrized with the value 1 to indicate that only one instance of each per `Token` is enabled. The attachment description in (line 23) indicates that the `iTimer` interface of the `ValidityChecker` component is bound to the `iTimer` interface of the `Timer` component.

7.3.1 Primitive components

For primitive component specification (see § 6.2.2), Figure 7.9 shows the Uppaal template modelling the `Timer` primitive component. The template can be read clockwise from the initial location distinguished with a double circle. The `Timer` waits for a `setTimeout` call with a parameter of type `TIME`[1] declared at the top of the channel label (`time:TIME`). When it receives such an event, it stores the time value in a local variable (`time:=t`), resets a clock variable `cl` to 0 (`cl:=0`) and goes

[1] `TIME` is a user defined data type

to the next location. Then, the `Timer` sends `E_..._setTimeout` indicating the end of the treatment of the `setTimeout` event (`setTimeout` is a synchronous event) and goes to the next location. This latter is decorated with an invariant (`cl<=time`) to indicate that the process should not stay at that location when the invariant becomes false (*i.e.,* `cl>time`). When that happens, the `Timer` enables the last transition by triggering a `timeout` event, resets the clock again to 0 (`cl:=0`), and returns to the initial location. Another possibility to leave the location with the invariant is to receive a `stopTimer` event to end a session due for example to a user logout request. In that case, the timer returns `E_..._stopTimer[time-cl]` indicating the end of the action and the remaining time for the user, it resets the clock to 0 and returns to the initial location.

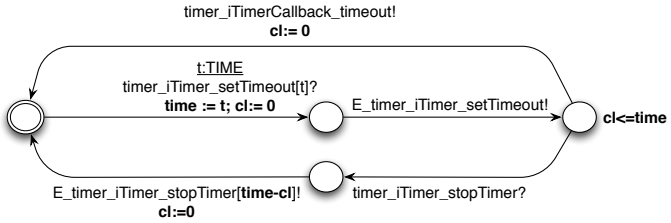

Figure 7.9: Formal Model for the `Timer` Component

Listing 7.2 shows the Uppaal textual description (XTA) of the `Timer` component. Data types (*e.g.,* `TIME`) are declared first (line 1). Each template in Uppaal is declared within the `process` keyword followed by the name of the template (`Timer` in this case) (line 2). Clock variables (`cl`) and local variables (`time`) are declared in the top of the declaration (line 3-4). Then the locations are listed (line 5), and the initial location is explicitly indicated (line 6). The transitions come last following the same syntax we adopted for behaviors specification in the ADL description (line 7-15).

```
1 typedef int [0,6]  TIME;
2 process Timer() {
3 clock cl ;
4 TIME time;
5 state l0 {cl<=time}, l1, l2;
6 init  l0;
7 trans
8     l0 -> l1 {select t:TIME; sync timer_iTimer_setTimeout[t]?; assign time:=t, cl:=0;},
9     l1 -> l2 {sync E_timer_iTimer_setTimeout!;},
10    l2 -> l0 {sync timer_iTimerCallback_timeout!;},
11    l2 -> l3 {sync timer_iTimer_stopTimer?;},
12    l3 -> l0 {sync E_timer_iTimerCallback_timeout[time-cl]!; assign cl:=0;};
13 }
```

Listing 7.2: The Uppaal-XTA description of the `Timer` component

7.3.2 Composite components

For composite component specification (see § 6.2.3), let us consider the `Token` composite component. This component has two interfaces: `iToken` and `iTokenCallback` (see Listing 7.1 line 19-20). The interface `iToken` has one synchronous service: `startToken` (Listing 7.1 line 4) for starting a new session. Figure 7.10 depicts the generated Uppaal template for `iToken` interface of the `Token` component (one cycle of four transitions). While the `iTokenCallBack` interface has a single asynchronous service: `timeout` (Listing 7.1 line 3) for signaling that the session time elapsed. Figure 7.11 depicts the generated Uppaal template modeling the `iTokenCallback` of the `Token` component (a single cycle of two transitions).

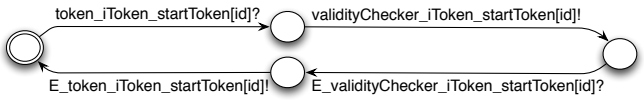

Figure 7.10: Formal Model for `Token`: `iToken` interface

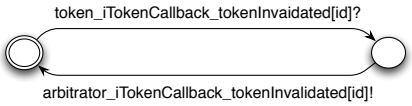

Figure 7.11: Formal Model for `Token`: `iTokenCallback` interface

7.3.3 Component binding

For component binding (see § 6.2.4), Figure 7.12 depicts the `Timer` template after binding its required interface `iTimerCallback` to that provided by the `ValidityChecker` component (Listing 7.1 line 23). Thus, the `timer_iTimerCallback_timeout!` is substituted by `validityChecker_iTimerCallback_timeout!`.

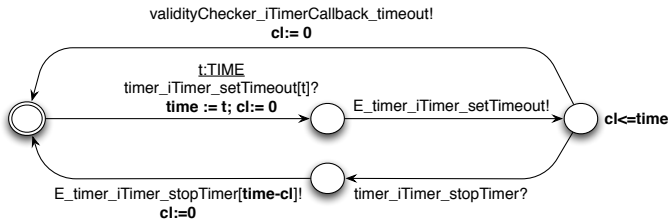

Figure 7.12: Formal Model for the `Timer` Component after binding

7.3.4 The complete base system

The whole example is modeled as 32 Uppaal template: 19 for primitive components (one for each component) and 13 for composite components (one for each bound interface). As described in Section 6.2.5. Primitive component behavior specifications are adapted to meet our notations for transition labels and component instantiation, composite components are automatically generated from the ADL specification and the binding is elaborated by renaming. In addition, aspects are also modeled as Uppaal templates (one for each aspect) and a weaving process is executed to synchronize the aspect process with the processes modelling the components on the required view.

Properties for the Airport Base System ($\mathcal{P}_{airport}$)	
Live 1	`User(id).Connected -> User(id).Disconnected`
Safe 1	`A[] not deadlock`
Safe 2	`A[] ∀(id:IDS),∀(ip:IPS) (User(id).Connected ∧ currentIp(id)==ip ∧ Firewall.enabled(id)) ⇒ User(id).isConnected`
Safe 3	`A[] ∀(id:IDS) User(id).Connected ⇒ User(id).cl<=validity(id)`
Reach 1	`E<> User(0).Connected ∧ (∀(id:IDS) id!=0 ⇒ User(id).Connected)`

Table 7.1: Properties of the airport system

The airport wireless access example is designed to satisfy different (liveness, safety, and reachability) properties. These are given in Table 7.1 ($\mathcal{P}_{airport}$). In particular, a user can not stay connected forever (*Live 1*), the system is deadlock free (*Safe 1*), a user cannot stay connected more than the validity time indicated in his flight ticket (*Safe 2*), a user can connect to all the IP addresses when its access is enabled by the firewall (*Safe 3*), and several users can be connected at the same time (*Reach 1*). The formulae rely on different constants, variables and auxiliary functions: `IDS` and `IPS` denote the range for user identifiers and IP addresses, respectively, `Connected` and `Disconnected` are identifiers denoting particular locations of the user process, the `validity(id)` is a global function that returns the authorized connection time of a user `id`, `currentIp(id)` returns the current IP address the user wants to connect, and `enabled(id)` checks whether a user id access is enabled by the firewall. The `cl` is a local clock associated to the user process, and the `isConnected` is a local variable in the user process that stores the firewall response of the user access to each IP address. These properties are later used for interference's detection.

7.3.5 Weaving individual aspects to the system

In this section we show how the desired aspects are woven individually to the system. At this step, the conformance of each aspect with the base system can be checked. The conformance is detected if the woven system satisfies the intrinsic properties of the base system and the intent properties of the woven aspect.

7.3.5.1 Weaving the `Bonus` aspect

The `Bonus` aspect is designed to satisfy a safety property *Safe 3'*: a user can stay connected a bonus time (`BonusTime`) after its authorized time elapses, where `BonusTime` is a constant denoting the bonus time associated to the user.

Safe 3' = `A[] ∀(id:ID) User(id).Connected ⇒ User(id).cl<=validity(id) + BonusTime`

For weaving the `Bonus` aspect (see § 6.2.6), the process modelling the `ValidityChecker` (the component owning the provided interface intercepted by the aspect) is instrumented by adding new locations and transitions in order to synchronize with the `Bonus` process. Figure 7.13 shows an excerpt of the `ValidityChecker` adapted to be synchronized with the `Bonus` aspect.

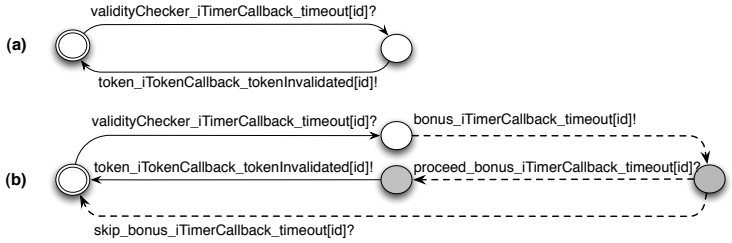

Figure 7.13: Synchronizing the `ValidityChecker` process with the `Bonus`: (a) before synchronization, (b) synchronized process

After instrumenting the `Bonus` aspect, the woven system is the parallel composition of the base system processes with the instrumented `ValidityChecker` process and the process modeling the `Bonus` aspect. Applying the `Bonus` aspect, we want to preserve all the basic system properties but *Safe 3* since the `Bonus` aspect is designed to offer a bonus time to users in addition to its original authorized time. In that case, $\mathcal{P}^{bonus}_{airport}$ is defined to be all the properties $\mathcal{P}_{airport}$ except *Safe 3*. Uppaal model checker shows that $\mathcal{P}^{bonus}_{airport}$ and $\mathcal{P}_{bonus}$ (*i.e., Safe 3'*) are all satisfied by the woven system with `Bonus` and no error is reported.

7.3.5.2 Weaving the `Alert` aspect

The `Alert` aspect intent is to ensure that a user is always alerted before it is disconnected (*Live 2*) and the alert is intercepted exactly before a `TimeAlert` of its elapsing time (*Safe 4*). In the formulae below, `Alerted` is an identifier denoting a particular location in the user process, `isAlerted` is a local boolean variable of the user indicating whether a user reached the `Alerted` location, and `AlertTime` is a constant denoting the alert time (*e.g.,* 5 minutes).

Live 2 = `Use(id).Connected --> User(id).Disconnected ∧ User(id).isAlerted`
Safe 4 = `A[] ∀(id:ID) User(id).Alerted ⇒ User.cl== validity(id) - AlertTime`

Weaving the `Alert` aspect requires the synchronization of the `ValidityChecker` process, having the provided interface intercepted by the aspect, with the process modeling the `Alert`. Figure 7.14 shows the `ValidityChecker` process synchronized with the `Alert`.

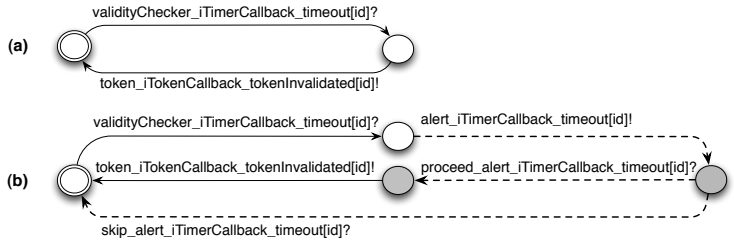

Figure 7.14: Synchronizing the `ValidityChecker` process with the `Alert`: (a) before synchronization, (b) synchronized process

Finally, the instrumented process with the `Alert` aspect instantiated for the intercepted join points are composed in parallel with the rest of the processes of the base system. Similar to the case of `Bonus`, Uppaal model checker reports no violation of base system properties and the aspect intent properties $\mathcal{P}_{alert}$ (*i.e., Live 2* and *Safe 4*). Note that in the case of `Alert`: $\mathcal{P}^{alert}_{airport} = \mathcal{P}_{airport}$.

7.3.5.3 Weaving the `NetOverloading` aspect

The intent of the `NetOverloading` aspect is to prevent access to P2P addresses for all the users when the airport server is overloaded. This property can be described in CTL as follows:

Safe 5 = `A[] ∀(id:ID), ∀(ip:IP) User(id).Connected`
`∧ currentIp(id)==ip`
`∧ NetOverloading.isP2P(ip)`
`∧ NetOverloading.isOverload() ⇒ !User(id).isConnected`

The above formula states that if the system is overloaded (`NetOverloading.isOverload()`) and the requested IP connection is P2P (`NetOverloading.isP2P(ip)`) then the user request is refused (`!User(id).isConnected`). In the formula, `isOverload()` and `isP2P(ip)` are predicates defined for the `NetOverloading` aspect process.

Weaving the `NetOverloading` aspect requires the adaptation of the `TIpDbConnection`, `PIpDbConnection`, and `Firewall` component processes. The adaptation enables the synchronization of the component processes with the aspect process. Figure 7.15 depicts the adapted `Firewall` process, where the added states are represented with dark circles and the added transitions are represented with dashed arrows.

The `NetOverloading` aspect is designed to disable the *Safe 2* property that checks whether a user has the access to all its requested addresses where he/she is connecting. Thus, $\mathcal{P}^{netOverloading}_{airport}$ is defined as $\mathcal{P}_{airport}$ except *Safe 2*. The parallel composition of the `NetOverloading` aspect process, the adapted component processes and the rest of the base system component processes reports no violation of

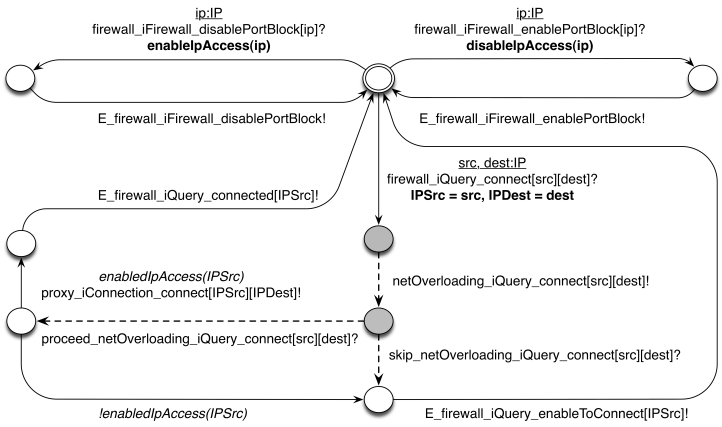

Figure 7.15: Synchronizing the `Firewall` process with the `NetOverloading` process

$\mathcal{P}_{airport}^{netOverloading}$ and $\mathcal{P}_{netOverloading}$ (*i.e., Safe 5*) which indicates that the weaving of the `NetOverloading` aspect is correct w.r.t to the airport base system.

7.3.5.4 Weaving the `LimitedAccess` aspect

The intent of the `LimitedAccess` aspect is to prevent access to blacklisted addresses for minors and enforces the access to all the requested addresses for first class users. These two properties can be expressed in CTL as follows:

$$
\begin{aligned}
\textit{Safe 5} \quad = \quad & \texttt{A[] } \forall\texttt{(id:ID), } \forall\texttt{(ip:IP) User(id).Connected} \\
& \qquad \wedge\ \texttt{currentIp(id)==ip} \\
& \qquad \wedge\ \texttt{LimitedAccess.isMinor(id)} \\
& \qquad \wedge\ \texttt{LimitedAccess.inBlackList(ip)} \Rightarrow \texttt{!User(id).isConnected} \\
\textit{Safe 6} \quad = \quad & \texttt{A[] } \forall\texttt{(id:ID), } \forall\texttt{(ip:IP) User(id).Connected} \\
& \qquad \wedge\ \texttt{currentIp(id)==ip} \\
& \qquad \wedge\ \texttt{LimitedAccess.isFirstClass(id)} \Rightarrow \texttt{User(id).isConnected}
\end{aligned}
$$

Safe 5 states that if a user is minor (`LimitedAccess.isMinor(id)`) and the requested IP connection is in the black list (`LimitedAccess.inBlackList(ip)`) then that user request is refused (`!User(id).isConnected`). *Safe 6* states that if a user has a first class ticket (`LimitedAccess.isFirstClass(id)`) then all that user requests must be accepted (`User(id).isConnected`). Where `isMinor(id)`, `isFirstClass(id)`, and `inBlackList(ip)` are predicates defined in the `LimitedAccess` aspect process.

Weaving the `LimitedAccess` aspect requires the adaptation of the `Arbitrator` and the `Firewall` component processes for their synchronization with the aspect process. Similar to the case of the `NetOverloading` aspect, the $\mathcal{P}_{airport}^{netOverloading}$ is defined as $\mathcal{P}_{airport}$ except *Safe 2*. When the processes are adapted and composed with the instantiated aspect process and the rest processes of the system, Uppaal model checker reports no error which indicates the `LimitedAccess` aspect is correct with respect to the airport base system.

7.3.5.5 Weaving the `Safety` aspect

The intent of the `Safety` aspect is to end user sessions when their flights are taking off. This can be expressed in CTL as:

$$\textit{Safe 7} \quad = \quad \texttt{A[] } \forall\texttt{(id:ID), User(id).Connected} \\ \wedge\ \texttt{Safety.flightTakeOffTime(id)>=gcl} \Rightarrow \texttt{!User(id).isConnected}$$

In the above formula, `flightTakeOffTime(id)` is a local function to the `Safety` aspect that returns the take off time of the `id` user plane and the `gcl` is a global clock defined for the system. Weaving the `Safety` aspect, the `Arbitrator` and the `FlyTicketClassifier` component processes must be adapted to be synchronized with the `Safety` aspect. In addition, $\mathcal{P}^{safety}_{airport}$ is defined as $\mathcal{P}_{airport}$, since the aspect is not designed to alter the basic behavior of the system but to enforce a new constraint. Uppaal also in this case does not report any violation.

Since The base system is proven that it is well-defined (*i.e.,* it satisfies all its intrinsic properties), all the desired aspects are correct with respect to the base system, let us check for potential interferences among the aspects.

7.4 Interference Detection and Resolution

In this section we discuss interferences among the above defined aspects. We use our formal model to detect such interferences. For each two aspects we introduce a sequence diagram of their composition and we show how they interfere with each other. Then we define the intent of each aspect as CTL formulae and we use Uppaal model checker to detect interferences and report a trace that violates the desired properties. Remember that, in our approach, when two aspects intercept common join points (*i.e.,* services), the advices of the two aspects on those services are executed sequentially, and the service call continue its original path (*i.e.,* proceeded) only if at least one of its underlying aspects decides to proceed the call, otherwise, the call is skipped.

7.4.1 `Bonus` vs `Alert`

Let us consider the original airport system with the bonus and the alert aspects. Let us also assume that the original session duration is 60 minutes, the bonus adds 10 minutes and the alert warns the user 5 minutes before the end of the session. When both aspects are bound to the system, we wish users to get a bonus time and be alerted exactly 5 minutes before the actual end of their sessions (*i.e.,* alert at 65 minutes, end of session at 70 minutes).

Figure 7.16 details the execution of the `Bonus` and the `Alert` behaviors sequentially. At time 0, a user logs in for 60 minutes provided by his flight ticket. The message `setTimeout(60)` sent from a `ValidityChecker` to its `Timer` is intercepted and forwarded to the `Alert` aspect only (*i.e.,* the `setTimeout(60)` is not a common intercepted call). The `Alert` subtracts 5 minutes from 60 and proceeds the call with

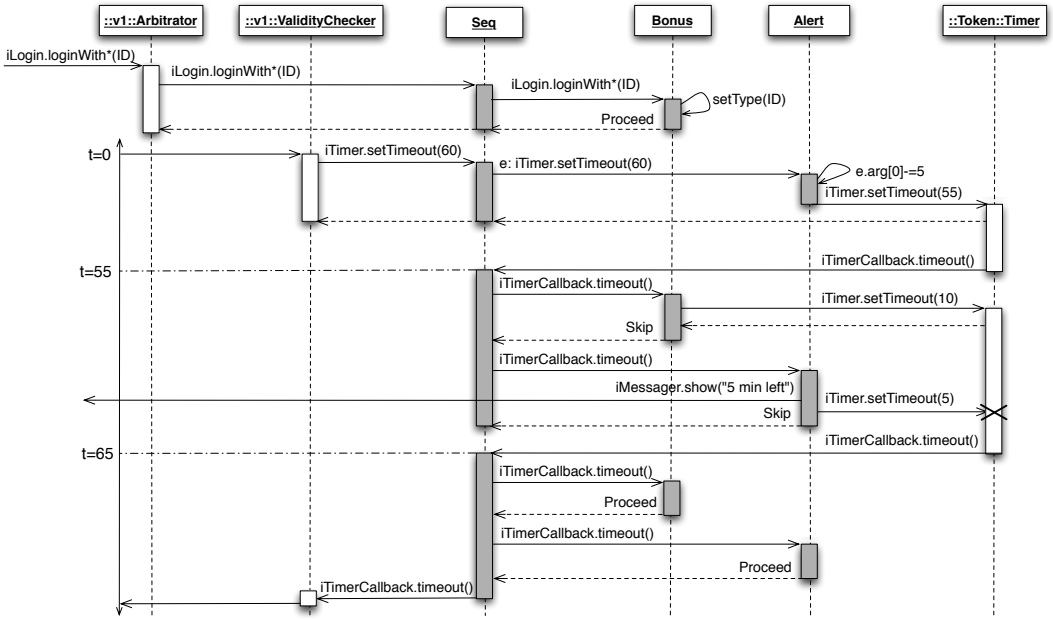

Figure 7.16: `Seq(Bonus,Alert)` scenario

the new parameter value (55). As a result, a `timeout` service call is intercepted at time 55. The `timeout` is common intercepted service so that its call is forwarded to the `Bonus` first then to the `Alert`. The `Bonus` resets the timer for 10 minutes and skips the call. Then, the call is sent to `Alert` that warns the user, resets the timer for 5 minutes and skips the call. This violates the expected behavior: the alert is sent too early (at time 55 instead of the expected 65). Moreover, the `Timer` has been set twice with different values and hence it is inconsistent whatever happens next. This is called an interference since the desired behavior is not satisfied by the default sequential composition of aspects. To solve this interference, another composition strategy is needed: *the first occurrence of* ***`timeout`*** *should only be managed by* ***`Bonus`*** *and the second occurrence should only be managed by* ***`Alert`***. Within this alternate strategy, when a service common to both aspects is intercepted, its occurrences are passed alternately to the left and the right hand side aspects. When a service is not common to both aspects, the call is forwarded to its corresponding aspect only.

Figure 7.17 details the alternate execution of the `Bonus` and the `Alert` scenario. At time 0, a user logs in for 60 minutes provided by his flight ticket. The message `setTimeout(60)` sent from a `ValidityChecker` to its `Timer` is intercepted and forwarded to `Alert`. The `Alert` subtracts 5 minutes from 60 and proceeds the call. As a result the `setTimeout(55)` call is proceeded. At time 55, a `timeout` is intercepted which is a common intercepted message by both aspects. Here, the call is forwarded only to `Bonus` (first occurrence) that resets the timer for 10 minutes and skips the message. Thus, the first `timeout` call is ignored. At time 65, a second `timeout`

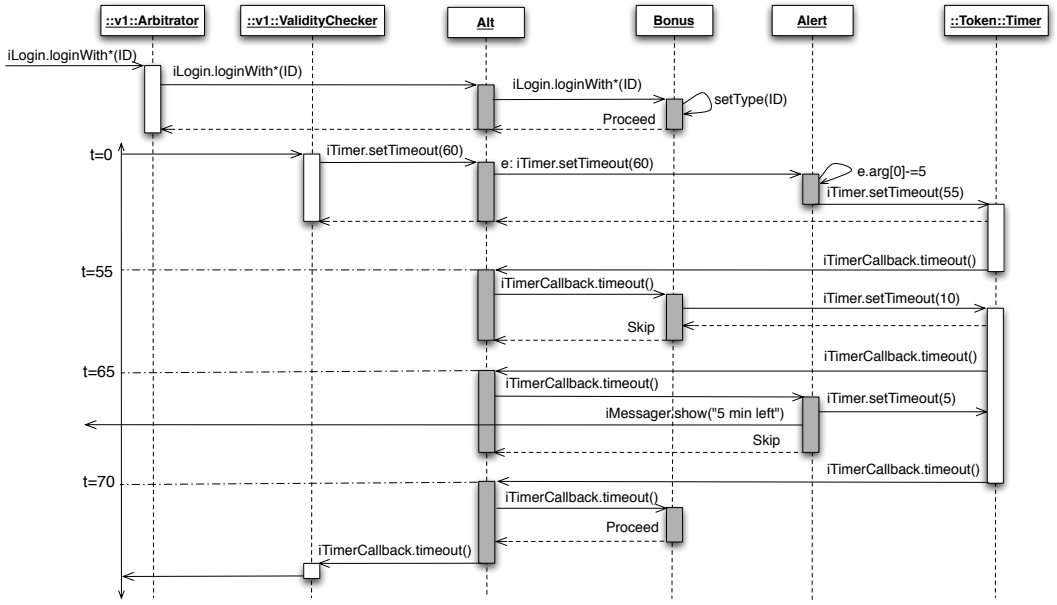

Figure 7.17: `Alt(Bonus,Alert)` scenario

is intercepted. The call, this time, is forwarded only to `Alert` (second occurrence) that warns the user, resets the timer for 5 minutes and skips the message. Thus, the `timeout` call is ignored again. At time 70, a third `timeout` call is intercepted and forwarded only to `Bonus` (third occurrence) that proceeds the call since the bonus time is consumed. As a result, the `timeout` is proceeded. This ends the session and elaborates the desired behavior.

When both `Bonus` and `Alert` are woven to the system and composed using `Seq` (see Figure 7.16), the Uppaal model checker reports that *Safe 3* property is violated and it gives a diagnostic trace similar to the sequence diagram in Figure 7.16. The reported trace shows that both aspects reset the `Timer` for two different values and the `Timer` is not designed to accept such kind of behavior. In addition, the alert message is send to the `User` before the addition of bonus. To solve the problem, we used the `Alt` operator (see Figure 7.17) that sets the `Timer` once for each intercepted `timeout` event, and ensures that the alert is sent to the `User` after consuming the bonus. The use of Uppaal model checker this time shows that all the desired properties are satisfied which indicates that the interference is solved.

For the implementation point of view, the original Fractal configuration is updated so that, a new composite modelling the view named `SessionManager` is created as a child of the root component (*i.e.,* the closest common parent of the required components for the view in the example), the `Arbitrator` and the `ValidityChecker` components are declared as shared components of the `SessionManager`, an interceptor controller is added to the `Arbitrator` and the `ValidityChecker` to intercept the

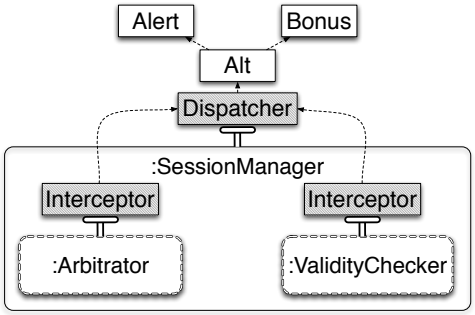

Figure 7.18: `SessionManager` view for `Alt` and `Bonus` aspects

required service calls and forward them to the composable controller `(Dispatcher, Alt(Alert,Bonus))` at the top of the `SessionManager` component. Figure 7.18 shows the structure of the `SessionManager` component. In the figure, shared components are depicted with double dashed lines, controllers are depicted with dark rectangles while `ICController`(s) are depicted with white controllers. The arrows linking controllers show the control flow of intercepted calls.

7.4.2 `LimitedAccess` vs `NetOverloading`

Let us consider the `LimitdtedAccess` and the `NetOverloading` aspects. While the former blocks access to blacklisted websites, which can be P2P as well, for minors and provides full access for first class customers, the latter blocks all access to P2P addresses for all the users when the airport server is overloaded. This results on an interference when the server is overloaded and a first class customer request to access to a P2P. Both aspects share a join point (*i.e.,* `IQuery` interface of the `Firewall`) and hence they are composed using the `Seq` operator. However, this default composition reports an error by Uppaal: *Safe 4* is violated. The reported diagnosis trace (see Figure 7.19) shows that the default sequential execution of such aspects does not solve the problem if the requested IP address by a first class customer is P2P address and the system is overloaded.

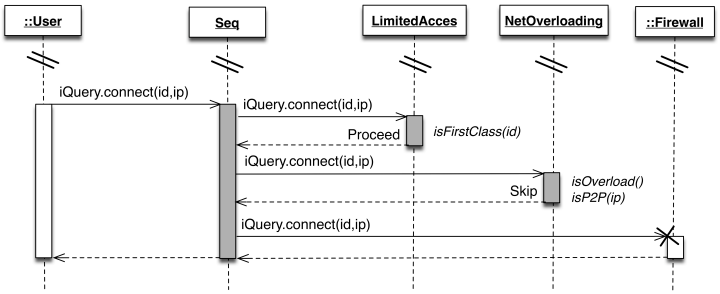

Figure 7.19: `Seq(LimitedAccess,NetOverloading)` scenario

For solving such an interference we must choose which aspect is prioritized. In other words, when the system is overloaded, either we do not accept P2P access for all users even for users owning first class tickets, or we block P2P access only for first class minor customers. In the former case, we use `And(LimitedAccess,NetOverloading)` operator (see Figure 7.20), while in the latter case we use `Cond(LimitedAccess,NetOverloading, "isMinor(id)")` (see Figure 7.21).

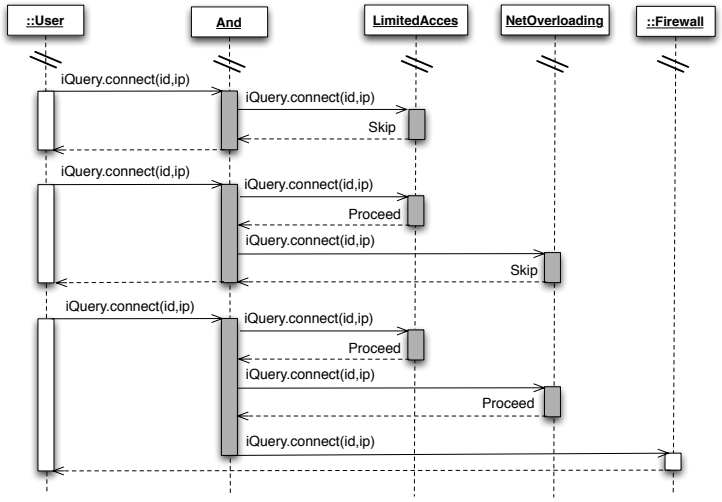

Figure 7.20: `And(LimitedAccess,NetOverloading)` scenario

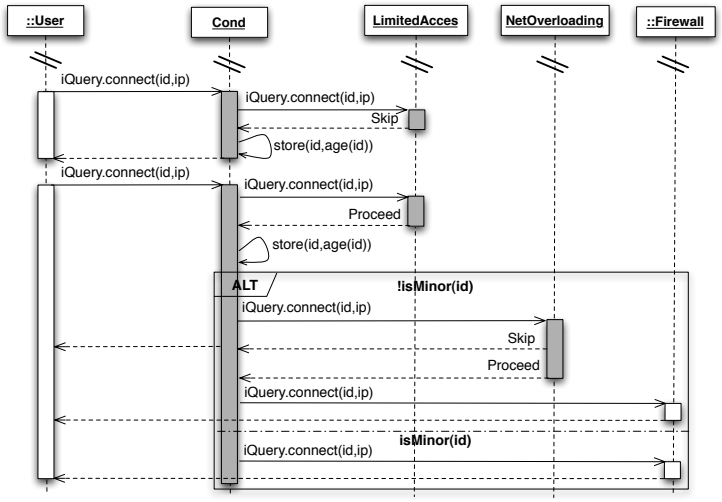

Figure 7.21: `Cond(LimitedAccess,NetOverloading,isMinor)` scenario

The implementation of such aspects requires the configuration of the Fractal system. In our approach, to compose two composable controllers, both controllers should be integrated into the same view. The views required by both aspects share

the `Firewall` component. The required view for their composition is calculated using the following VIL expression:

$$vNetOverLoadingLimitedAccess = vLimitedAccess \oplus vNetOverloading$$

The result view encapsulates all the components defined by *vLimitedAccess* and *vNetOverloading* and the resulting controller from the composition of both controllers is added to that new view as shown in Figure 7.22.

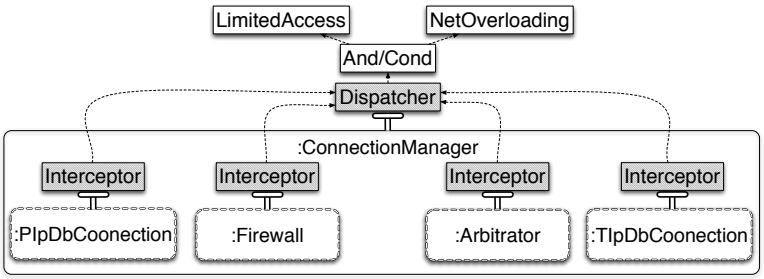

Figure 7.22: The view for `LimitedAccess` and `NetOverloading` aspects

7.4.3 `Safety` vs `Alert` and `Bonus`

According to our approach, since both aspects do not share join points, there is no need to compose them using `Seq`, instead, they are executed in parallel. However, this may also lead to an undesirable behavior. Take for example, the `Safety` aspect, its intent states that when a user `id` is connected and the plane of this user is taking off, the user disconnects immediately. Using Uppaal model checker, it shows that all the properties of `Alert` (*i.e., Live 2* and *Safe 4*) are violated (*i.e.,* the user is not alerted before five minutes of the end of the session). The reported trace (see Figure 7.23 where `Par` refers to the regular parallel composition of processes) shows that an interference among the `Safety` and the `Alert` aspects appears when the `Alert` already subtracts 5 min from the authorized time and the `Safety` wants to close the session because the flight of the user is taking-off. In that case, the `Alert` is waiting for the first timeout event to alert the user and reset the timer with 5 minutes. The `Safety` aspect is urgent and the session must be closed but this deprives the user of 5 min of his/her authorized time and prevents alerting the user. To solve this problem, one solution is to execute the `Safety` aspect only if the authorized time for a user is greater than the remaining time for the flight to take off, otherwise, both aspects can be executed without any interference (*i.e.,* use the variant of the `Cond` operator, `Cond(Safety,Alert,validity(id)>=takeOff(id))`). When the `Bonus` aspect is considered, neither `Alert` nor `Bonus` should be executed if the authorized time with bonus is greater than the remaining time to the taking off of the plane (*i.e.,* `Cond(Safety,Alt(Bonus,Alert),validity(id)+BonusTime>=takeOff(id))`).

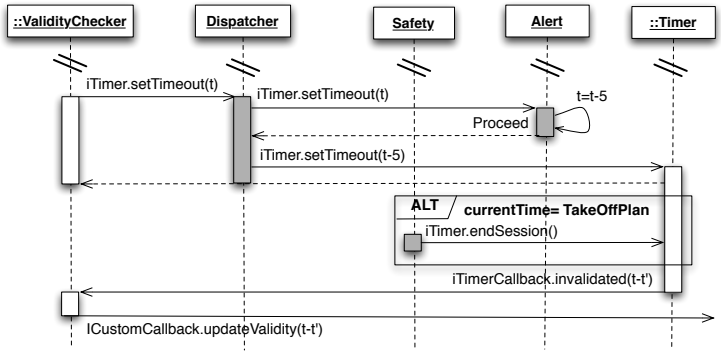

Figure 7.23: `Par(Safety,Alert)` scenario

	Verification time (sec)		
	1 users	**2 users**	**3 users**
Safe 1	0.03	1.84	220
Safe 2	0.01	1.21	141
Safe 3	0.02	1.31	167
Reach 1	0.01	0.11	3.23
Live 1	0.01	6.22	?
	System + Bonus		
Safe 3'	0.03	9.03	?
	System + Alert		
Safe 4	0.05	16.11	?
Live 2	0.11	154	?
	System + Bonus + Alert		
Safe 3'	0.14	43.11	?
Live 2	0.11	154	?
Nb of processes	32	38	44

Table 7.2: Verification Time for Checking Properties

7.5 Conclusion

In this chapter, we have exemplified the use of VIL language to define the required views for aspect weaving. In addition we have shown how interferences are detected using Uppaal model checker after formalizing the base system as a network of state machines, and how the interferences are solved using composition operators. The case study is modeled as 32 Uppaal templates in total. Table 7.2 shows the number of processes generated and the respond time of Uppaal model checker for each property. The table shows that Uppaal does not give respond for more than two users in a woven system for our case study. However, for our verification, merely one user is enough to detect interferences among the woven aspects. To avoid state explosion some abstractions have to be made. By abstraction, some composite components with their internals that are not part of views can be replaced by primitive processes abstracting the behavior of the composite.

Chapter 8

Conclusion

Contents

Aspect-oriented programming provides a support to deal with non-modular features of software systems ensuring their modularity and reusabibilty. CBSE provides a modular way to design software systems enhancing their reusability and reducing their maintenance effort. The idea behind AOP is to model a non-modular feature as one entity named aspect. An aspect consists of the definition of one or more methods, called advices, describing the feature behavior and an expressive language to describe points in the base program, named join points, specifying where advices must be executed. In addition, AOP provides a weaving mechanism that enables the execution of advices in the right places in the base program. On the other hand, CBSE divides a software system into separate entities, named components, and provides a mechanism to compose them to get the required complete system. However, like the other paradigms, CBSE suffers from modelling non-modular features that are scattered over several components and can not be modeled as regular entities. For the sake of ensuring better modularity and reusability of component software in CBSE, non-modular features must be added in a modular way, and hence, AOP aspects should be integrated into CBSE models. Achieving this aim, a modelization of aspects and their constituents (*i.e.,* join point description language, advices, and the weaving mechanism) in component models are required. In this thesis, we proposed a generic approach to model aspects. The proposed approach gets benefit of component configuration diversity to specify the adequate system configuration for each aspect to be added. However, adding one or more aspects may give rise to interferences where the execution of one aspect may alter the execution of another. For this reason, we provided a support to detect and solve aspect interferences in component models. In our proposal, aspects are modeled as wrappers on views. A view is an adequate configuration for a wrapper, it encapsulates all the components of interest of a wrapper in the same composite. Views are clearly visible in hierarchical component models such as Fractal, where composites are created to encapsulate the required components for a wrapper. In flat component models such as EJB, a view is abstract, it only defines the points where a wrapper must be added as a regular component bound to the components defined in the view. For declarative definition

of views, we introduced VIL: a declarative pointcut language adopted for component models. For interference detection and resolution, a formalization process is defined: First, an abstract ADL is proposed to describe the component architecture structure and bevavior. Both primitive components and aspects behavior is given in the ADL as state machines with a conventional notation of transitions labels. Second, a translation scheme is executed to generate the full formalization of the aspectualized component system from the ADL description, this includes the adaptation of primitive components processes, composite components processes generation, component binding and aspects weaving. The result system can be used as input to a model checker with the set of intrinsic properties of the system and the intent of each aspect. An interference is detected when at least one of the intent properties of aspects or the intrinsic properties of the system is violated. To solve interferences, a catalog of composition operators' patterns is provided. Our approach is exemplified with two case studies: a crane system and an airport wireless access. For each example, an implementation in Fractal component model is provided, a set of aspects are defined and woven to the system, and interferences are detected using Uppaal model checker and solved using adequate composition operators. The proposed approach is designed to be general (it does not rely on a specific component model), we have shown how views can be modeled and how wrappers can be implemented into two different categories of component models: hierarchical component models with component sharing such as Fractal and flat component models such as EJB.

8.1 Aspectualizing Component Models

There are several ways to extend component models with AOP support, each method depends on the underpinnings of each model. However, this requires an effort to extend each model. A unified approach is required to save those efforts. The concept of views proposed in this thesis is a generic concept that can be adapted for different component models. Moreover, classical weaving strategies (*i.e.,* compile time, load time and runtime) harms the modularity of components and disables dynamic weaving/unweaving of aspects. To tackle this problem, component wrapping/monitoring are considered for component models. This preserves the modularity of both components and aspects before and after weaving and enables dynamic weaving/unweaving of aspects by activating/deactivating wrappers. Our views-based idea is a promising approach to locate the scope of each aspect, views can be seen clearly in hierarchical component models (*i.e.,* they are modeled as composites), but in flat component models views are still abstract units where only the interesting points of an aspect are specified and dummy components or connectors are used to forward the intercepted calls into the components modelling aspects. Adding dummy components and connectors hardens and complicates the original architecture of systems. This has direct side effects on the pointcut expressions that are originally defined for a base system architecture that is changed after weaving other aspects. Pointcut expressions for each aspect must always be defined for the current system

architecture. Note that views can be defined at earlier stages of software development processes. For example, at the design stage if some non-functional properties are already known, the base system architecture will be already designed with views and wrappers implementing such non-functional features. In addition, views can be used at the maintenance stage where new non-functional requirements are required for an already designed and implemented system.

8.2 Aspect Interaction Analysis

When several aspects are woven to the same component system their interactions must carefully be analysed in order to avoid potential interferences among them that lead to misbehaving the complete system. However, interferences are not always avoidable by composition operators, it depends on the kind of their interactions and on their effects on the system. Aspects interact in different ways, our experience shows that three kinds of aspect interactions are available:

Neutral: Two aspects are neutral when they are woven to the same system the result system satisfies the intent properties of all the woven aspects in addition to the system invariant.

Positive Interaction: Two aspects interact positively when the execution of one aspect reinforces the execution of another by fulfilling the precondition of the second aspect.

Negative Interaction This appears when the weaving of an aspect individually to a system is correct but its weaving to an already woven system leads to undesirable behavior. This means that the execution of one aspect alters the execution of another.

Dealing with aspect interferences is a tedious work where aspect interactions must be carefully analysed. This can be achieved by specifying clearly what is intended by a component system before and after weaving aspects and the intended behavior of each aspect to be woven. This enables the detection of conflicting properties and contradictory ones which is an error prone task if it is done manually. Tools such as model checkers are useful to detect such conflicting and contradictory properties. Due to state explosion problem, the use of model checkers is still limited to component systems with no considerable number of components and/or an optimized behavior specification. In addition, our proposition of a catalog of composition operators helps users to choose the adequate composition strategy that solve detected interferences. An alternative solution is *aspect adaptations* where one aspect should be aware of the presence or the absence of other aspects so that it may decide what to do with each intercepted join point. However, aspect adaptation is hard and error prone task especially that conflicting aspects cannot be easily and statically determined. But when operators does not provide a solution component adaptation can be used.

8.3 Perspectives

Besides the generality of our approach, some limitations of the approach and how to be tackled are: (1) the use of model checkers to model huge component systems following our approach leads to state explosion. To tackle this problem, we adopted system abstractions where composite components with their internals not part of the view are replaced with a primitive process modelling the external behavior of such composites. However, this is hard and error prone task. One solution is to use theorem proving such as B method. Using B, each component can be modeled as a B machine with its behavior specification in terms of pre/post conditions for each service and a protocol to describe the control flow of each component. Thus, the B tool can be used to check the conformance of intrinsic properties of systems with the different intent properties of aspects to detect interferences. (2) the composition patterns catalog must be extended with more operators. This can be achieved by experimenting the approach with more concrete examples and other aspect interactions. (3) actually, views are defined for centralized systems, where aspects are defined for components in the same server, for distributed systems where components are deployed into different servers, distributed AOP features must be taken into account. (4) the approach is adopted for component models, where the basic architecture of the system is defined, but what about software architecture based models where aspects are considered at earlier stages of software design. One proposition is to adapt the approach for component-based model-driven software development such as rCos [Jifeng 2006, Chen 2008]. rCos is an ongoing project aims to support multiple dimensional modeling: models at different levels of abstraction related by refinement relations, and models of different views of the system (structural, behavioral, and non functional ones).

Appendix A

Résumé en Français

A.1 Introduction

La programmation par composants (CBSE) permet la modularisation des préoccupations en termes d'entités logiciels séparées appelées composants. Chaque composant fournissant explicitement des services en s'appuyant sur des services fournis par d'autres composants. Les composants peuvent être assemblés afin de construire le système global. D'autre part, l'approche aspects (AOP) vise à séparer les préoccupations techniques ou de contrôle (*e.g.,* synchronisation, persistance, contraintes temps réel, etc.) des préoccupations métier ou fonctionnelles. Elle offre un mécanisme de tissage qui permet de fusionner ces deux types de préoccupations afin de construire le système global. Ceci permet une meilleure séparation du code fonctionnel de celui de contrôle ou technique et d'assurer une meilleure maintenabilité du système. Les préoccupations transversales ne sont pas liées à un paradigme spécifique et le paradigme composants n'est pas une excéption. Malheureusement, les travaux actuels sur la programmation par composants visent à implanter les concepts d'AspectJ [Kiczales 2001b] tels quels dans les modèles à composants ignorant la particularité des composants et des systèmes à composants (*e.g.,* points de coupures définissant des points dans les architectures composants) et les interférences entre les aspects qui peuvent apparaître lorsque plusieurs aspects sont tissés à un système. En fait, la détection et la résolution des interférences d'aspects est toujours un défi pour AOP. Dans cette thèse, nous contribuons par l'introduction d'un langage déclaratif de points de coupure (VIL) dédié aux modèles à composants, et nous fournissons un cadre formel pour la détection et la résolution des interférences d'aspects lorsque plusieurs aspects sont tissés à un système à composants. Dans ce cadre, nous introduisons une extension aux ADL(s) actuels par une définition explicite des comportements des composants et des aspects, et des règles de tissage et de composition d'aspects. Chaque règle utilise des expressions VIL afin de décrire déclarativement où les aspects vont être tissés. Nous fournissons un ensemble de règles de transformation pour obtenir la spécification formelle des composants et des aspects à partir de l'ADL, et nous utilisons des "*model checkers*" pour la détection des interférences possibles entre les aspects. Pour la résolution des interférences, nous fournissons un ensemble d'opérateurs de composition. Chaque opérateur est donné avec un exemple de motivation, une structure et un ensemble de règles d'applicabilité. La structure d'opérateur est décrite comme une forme abstraite qui peut être instanciée pour n'importe quel aspect et n'importe quel ensemble de points de jointure. La liste des opérateurs forme une première étape vers un catalogue pour la résolution

d'interférences d'aspects. Dans notre proposition, nous adoptons l'utilisation du model checker Uppaal pour son soutien à l'instanciation des processus, la déclaration des variables locales, et le passage de paramètres entre les processus, en plus de son soutien à des contraintes temporelles pour modéliser les systèmes temps réel. Enfin, il convient de mentionner que notre approche est générale et peut être utilisée pour d'autres modèles à composants avec des adaptations minimales.

A.2 Background

Notre expérience montre qu'un modèle générique composant-aspect doit satisfaire un ensemble de critéres que nous citons ci-dessous afin de permettre une meilleure modularité, réusabilité et préservant l'encapsulation des composants et des aspects avant et après le tissage.

Hiérarchie: par hiérarchisation, les différents composants peuvent être encapsulés par un composant composite. À notre avis, un composite ne doit pas être seulement une abstraction et une encapsulation d'un ensemble de composants, il doit avoir son propre comportement vis-à-vis des appels entrants et sortants de ses interfaces externes. Par example, il peut décider de suivre l'appel ou le bloquer. Cela permet aux aspects de s'intégrer au niveau composite et d'être appliqués à tous leurs composants internes.

Langage de coupure approprié aux composants: un langage de points de coupure devrait être suffisamment expressif pour définir l'ensemble des points de jointure associés à un aspect. Les modèles actuels utilisent le même langage de point de coupure proposé pour les objets au monde des composants. Nous comprenons que cela est dû au fait que la plupart des modèles de composants sont basés sur des objets. Toutefois, les points de coupures spécifiques pour les composants permettent la définition des points de jointures de manière déclarative. Par exemple, l'extension de Fractal FAC [Pessemier 2006, Pessemier 2008] définit deux mots-clés: "`client`" et "`server`" pour indiquer l'ensemble des interfaces requises et fournies par un composant. Ce sont des points de coupures spécifiques aux composants qui sont utilisés au lieu de préciser explicitement une par une les différentes interfaces d'un composant à être interceptées.

Support pour la détection des interférences: la détection des interférences d'aspects est une caractéristique principale d'un modèle à composants utilisant des aspects pour modéliser ses propérietés transversales ou non modulaires. Sans ce support, il n'y a aucune garantie que les aspects tissés fonctionnent correctement. Le développeur d'un modèle à composants doit être guidé par ce support pour indiquer s'il y a lieu une interférence et éventuellement lui indiquer ce qu'il faut faire pour résoudre le conflit. En AspectLEDA [Navasa 2009] les interférences entre aspects sont détectés pendant la phase de conception et leur tissage nécessite la mise à jour des diagrammes de séquence UML du système de base; à ce stade les développeurs peuvent détecter les interférences

possibles manuellement en observant le diagramme résulat du tissage. Malheureusement, le diagramme de séquence UML ne permet que la détection d'interférences structurelles et non pas des d'interférences comportementales (*i.e.,* des interférences aux points de jointures partagés). Notez que la détection des interférences est toujours un problème en AOP.

Composition d'aspects: Le tissage de plusieurs aspects est un source d'inconsistance du système. Afin de pallier ce problème, une solution est d'offrir un mécanisme de composition d'aspects qui nous permette de spécifier quels sont les aspects qui doivent être exécutés et sur quel ordre pour chaque point de jointure. En plus, nous pensons que les model checkers peuvent aider à la détection des interférences et donner suffisamment d'informations sur les aspects et leurs propriétés violées. Nous croyons également qu'une bibliothèque des patrons des opérateurs de composition peut aider à déterminer la bonne stratégie de composition pour résoudre les interférences qui apparaissent. AspectCCM [Clemente 2002], AES [Choi 2000] et SpringAOP [Pawlak 2005] permettent de déterminer l'ordre d'exécution des aspects aux points de jointure partagés, mais dans certains cas, l'ordre ne devrait pas être le même sur tous les points de jointure. Pour surmonter ce problème, AES [Choi 2000], CAM/DAOP [Pinto 2003, Pinto 2005], JBossAOP [Pawlak 2005] et AspectLEDA [Navasa 2009] permettent de déterminer et de spécifier l'ordre d'exécution au niveau advices. Cependant, ces deux niveaux de composition sont nécessaires et doivent être considérés ensemble. Par exemple, lorsque deux aspects partagent plusieurs points de jointure et l'ordre d'exécution des advices est le même pour tous les points de jointure, la composition au niveau aspects est utile, autrement, la composition au niveau advices est nécessaire. Cependant, la composition d'aspects va au-delà de l'ordonnancement, en particulier lorsque les interférences sont dues au tissage des aspects sans points de jointure partagés. JAsCo [Suvée 2003] supporte à la fois l'ordonnancement au niveau aspects et au niveau advices, et il fournit également une manière programmatique pour développer des stratégies de composition plus complexe.

Modularité d'aspects: la modularité d'aspects est la possibilité de définir le comportement de l'aspect dans un module séparé et préserver cette modularité après le tissage. La plupart des modèles à composants considèrent les aspects comme des entités de première classe (à savoir, des composants, des contrôleurs, des connecteurs, etc.), mais la préservation de cette modularité après le tissage dépend fortement de la stratégie de tissage de chaque modèle. Le tissage montre comment les aspects sont intégrés dans les composants. En AOP deux stratégies majeures de tissage sont fournies: statique et dynamique. Dans le monde des composants, le tissage statique n'est pas toujours possible car les composants sont souvent disponibles sous forme binaire. D'autre part, le tissage dynamique est techniquement plus complexe que le tissage statique. En outre, le tissage dynamique n'est pas possible pour les composants cryptés ou signés numériquement. Cela montre que ces deux modes de tissage sont

inutiles dans le monde des composants. PRISMA [Pérez 2006, Pérez 2008], AspectLEDA [Navasa 2009], et SpringAOP [Pawlak 2005] utilisent le tissage statique des aspects et par conséquent la modularité des aspects n'est pas conservée après le tissage par ces modèles. Dans le monde des composants, le wrapping ou le monitorat est la stratégie qui peut être utilisée quel que soit le type de composants et permet le tissage et le détissage d'aspects selon les besoins. Par wrapping, les aspects devraient être modélisés comme des entités de premier ordre et les composants sous-jacents devraient être surveillés de sorte que lorsque des points de jointure sont atteints les aspects sont appelés.

Instanciation d'aspects: plusieurs instances d'un aspect peuvent apparaître dans le même système. Ceci est important quand un aspect définit une ou plusieurs variables d'état qui sont mises à jour quand certains points de jointure d'un composant sont atteints. Dans ce cas, tous les composants pertinents d'un aspect sont contrôlés par une seule instance et les variables d'état de l'aspect sont mises à jour uniquement pour les composants définis. La plupart des modèles à composants prennent en charge l'instanciation d'aspects. Cependant, les modèles à base des conteneurs tels que CCM [OMG 2006] et EJB [Burke 2006], ne donnent aucune information sur l'instanciation d'aspect soit parce que les aspects pris en charge sont singleton ou que l'instanciation est automatiquement gérée par leurs conteneurs.

Le tableau A.1 présente un résumé et une comparaison directe des différents modèles à composants. La comparaison est basée sur l'ensemble des critères d'AOP présentés ci-dessus. Comme indiqué dans le tableau, aucun des modèles étudiés ne fournit un support complet de tous les critères, ce qui motive fortement le travail de cette thèse. De notre point de vue, un modèle d'aspects complet doit être générique (applicable à différents modèles à composants) et offrir un support efficace des aspects dans les modèles à composants (instanciation d'aspects, détection des interférences entre les aspects et la composition des aspects).

	Hiérarchie	Langage approprié de PC	Détection Interf.	Niveau de Composition		Modularité		Inst.
				Aspect	Advice	Avant	Après	
EJB	-	-	-	-	-	+	+	?
AES	-	-	-	-	precedence	+	-	+
CCM	-	-	-	-	-	+	+	?
AspectCCM	-	-	-	precedence	-	+	+	+
Fractal	+	-	-	-	-	+	-	+
Safran	+	-	-	-	-	+	+	+
Fractal-AOP	+	-	-	-	-	+	+	+
FAC	+	CLIENT, SERVER	-	precedence	-	+	+	+
JAsCo	-	-	-	+	+	+	+	+
CAM/DAOP	-	-	-	-	precedence	+	+	+
Spring AOP	-	-	-	precedence	-	+	-	+
JBoss AOP	-	-	-	-	precedence	+	-	+
PRISMA	+	-	-	+	-	+	-	+
AspectLEDA	-	-	UML seq. diagram	-	precedence	+	-	+

Table A.1: Le modèles à composents et leurs niveaux de support des aspects: (+) un support complet de la propriété, (-) pas de support de la propriété, (?) inconnu

A.3 Les aspects et les vues

Dans le processus de développement des systèmes à composants, les configurations des systèmes sont définis par des architectes de logiciels qui décident quels composants sont utilisés et comment ils sont connectés. Les architectes de logiciels, selon la spécification des besoins, choisissent la configuration appropriée (assemblage) aux systèmes. En pratique, pour un système donné, différentes configurations sont possibles, chacune répond à une ou plusieurs exigences. Partant de ce constat, nous utilisons le terme vue pour référer une configuration du système adoptée à une ou plusieurs exigences (système de base). En outre, nous utilisons le terme wrapper pour désigner une entité qui encapsule un composant, intercepte ses appels entrants et/ou sortants, exécute un extra-code, et explicitement poursuit ou pas les appels d'origine (un aspect). Dans ce qui suit, on présente le langage de points de coupure VIL que l'on propose pour les modèles à composants.

A.3.1 Le Langage VIL

VIL est un langage de définition des vues qui permet de définir des configurations des systèmes à composants requises pour le tissage des aspects. Une expression VIL spécifie à la fois l'ensemble des composants à encapsuler et l'ensemble de leurs interfaces et services qui doivent être interceptées.

$$
\begin{array}{lll}
vexp \in View & ::= & * \\
& | & cId \\
& | & \textbf{child}\ [^{+}]\ v \\
& | & \textbf{parent}\ [^{+}]\ v \\
& | & \textbf{primitive}\ v \\
& | & \textbf{instance}\ v \\
& | & [\textbf{direct}]\ \textbf{provide}\ v\ [(\textbf{T} \mid \textbf{N})\ id^{*}][sig^{*}] \\
& | & [\textbf{direct}]\ \textbf{require}\ \ v\ [(\textbf{T} \mid \textbf{N})\ id^{*}][sig^{*}] \\
& | & \textbf{bound}\ [\textbf{C} \mid \textbf{S}]\ v\ [id^{*}] \\
& | & \textbf{attributes}\ v\ [atId^{*}] \\
& | & \textbf{scflow}\ v \\
& | & v_1 \oplus v_2 \\
& | & v_1 \otimes v_2 \\
& | & v_1 \ominus v_2
\end{array}
$$

Dans notre langage, présenté ci-dessus, "$*$" indique que tous les composants de l'architecture doivent être encapsulés en un seul composant et toutes leurs interfaces et services sont interceptés. Ainsi, la vue requise est la même que celle d'origine (*i.e.,* pas besoin de reconfigurer le système) et le wrapper, dans ce cas, doit être appliqué au composant racine de l'architecture. *cId* se réfère à un identificateur de composants, ce qui indique que la vue requise est celle d'origine et toutes les interfaces du *cId* sont interceptés, et le wrapper est ajouté au composant nommé *cId*. Les mots-clés **parent** et **child** indiquent que la vue doit encapsuler tous les composants internes (resp. parents) de v dans le même composant et toutes leurs interfaces sont interceptées."+" dénote une fermeture transitive des relations ci-dessus. **primitive** v désigne une vue où tous les composants primitifs de v doivent

être dans le même composite et toutes leurs interfaces sont interceptés. **instance** v désigne une vue où toutes les instances de v doivent être wrappées et toutes leurs interfaces sont interceptées. **require** v et **provide** v désignent la même vue que v, mais seules les interfaces requises (résp. offertes) sont interceptées. Pour une spécification plus exhaustive, nous notons le nom (**N**), le type (**T**) des interfaces à être interceptées et potentiellement les signatures de services à saisir (*sig*). **bound** v désigne l'ensemble des composants liés aux composants de v. Pour être plus précis, nous fournissons (**C**) et (**S**) pour désigner uniquement les composants liés à une interface requise ou offerte de v, respectivement. Un motif d'identifiant peut également être utilisé pour sélectionner uniquement les composants dont les noms correspondent à un motif. **attribute** v désigne une vue qui encapsule tous les composants de v et intercepte tout accès à l'ensemble ou à certains de leurs attributs. Nous utilisons **scflow** v pour définir une vue qui encapsule tous les composants dans le flux de contrôle statique de v. Enfin, les vues peuvent être composées en utilisant les opérateurs : union ($\oplus$), intersection ($\otimes$) et différence ($\ominus$).

A.3.2 VIL en Fractal

Le modèle à composant Fractal [Bruneton 2004] définit son propre langage de description d'architecture (ADL) pour décrire la configuration des systèmes à composants. Il supporte la hiérarchie des composants, les contrôleurs pour les composants, l'introspection et le partage des composants. Les contrôleurs Fractal interceptent les appels entrants/sortants des interfaces des composants et permettent l'adaptation des comportements des composants et la définition des propriétés extra-fonctionnelles aux composants par l'exécution d'un code supplémentaire lorsque les appels sont interceptés. Nous bénéficions des contrôleurs Fractal pour définir les aspects. Cependant, dans la mise en œuvre de Fractal Julia, quand un composant possède plusieurs contrôleurs, il n'y a pas une manière générale de les composer. Ils ne peuvent être qu'exécutés indépendamment ou de façon séquentielle en configurant les intercepteurs, ou bien être composés d'une façon programmatique en appelant explicitement un contrôleur depuis un autre. Cela rend la mise en œuvre des contrôleurs une tâche complexe avec un comportement parfois inattendu. Pour faire face à cette limitation, nous introduisons les contrôleurs composables dans la mise en œuvre de Fractal Julia.

Les contrôleurs composables, comme leur nom l'indique, ont les mêmes propriétés que les contrôleurs Fractal réguliers (font partie des membranes des composants, leur rôle est de contrôler le comportement des composants et introduire des propriétés extra-fonctionnelles) mais ils peuvent être composés avec d'autre contrôleurs à l'aide des opérateurs de composition. Dans la suite, nous introduisons le concept de contrôleurs composables, nous décrivons comment les vues sont définies en utilisant VIL, et nous montrons comment les aspects ou les contrôleurs peuvent être composées pour résoudre les interférences possibles.

A.3.2.1 Les contrôleurs composable

Nous définissons un contrôleur composable comme une paire `(Dispatcher,ICController)` où `Dispatcher` est un contrôleur Fractal régulier ajouté à la membrane d'un composant et le `ICController` est un objet Java qui implémente l'interface *ICController* (voir Listing A.1). Le rôle du contrôleur `Dispatcher` est d'intercepter les appels vers les composants intérieurs de la vue, réifier les appels interceptés en un objet *MessageContext* et appeler la méthode *match()* de l'`ICController`. Le `Dispatcher` attend la décision prise par le `ICController` (*proceed* ou *skip*) et se comporte en conséquence. Quand il reçoit proceed, il fait appel à la méthode d'origine et l'appel atteint sa cible, quand il reçoit skip le `Dispatcher` ne fait rien et l'appel est ignoré. Pour composer des contrôleurs, un ensemble d'opérateurs de composition sont définis comme `ICController`(s), cela permet la composition des contrôleurs sous la forme de patron composite. La méthode *match(MessageContext)* de chaque opérateur de composition met en œuvre la sémantique associée à l'opérateur et retourne proceed ou skip au contrôleur `Dispatcher`.

```
1 enum Action {Proceed, Skip}
2 interface ICController {
3   Action match(MessageContext c);
4 }
```

Listing A.1: L'interface `ICController`

Généralisons et détaillons la mise en œuvre de vues pour Fractal. Pour la définition des vues en Fractal Julia, nous distinguons deux cas: (1) la vue courante est la même que la vue désirée (les composants à être wrappés appartiennent au même composite), et (2) la vue courante n'est pas la même que la vue désirée (les composants à être wrappés n'appartiennent pas au même composite). Dans la suite nous détaillons comment les vues sont créées pour chaque cas.

A.3.2.2 Cas 1. Vue courante = Vue désirée:

Dans ce cas, la vue requise est la même que la configuration d'origine. En pratique, cela nécessite de mettre à jour la description ADL de l'architecture (1) en déclarant un contrôleur composable modélisant l'aspect comme une partie de la membrane du composite encapsulant les composants requis et (2) en déclarant un contrôleur d'interception dans la membrane de chaque composant requis. En plus, un fichier de configuration Julia est généré pour inclure les nouvelles structures des membranes des composants. Le listing A.2 montre la modification générale nécessaire de la description ADL où le composant nommé *c* modélise la vue requise. Dans ce cas, un contrôleur composable est défini pour ce composant (`CControllerName` ligne 12), et un contrôleur intercepteur est défini pour chacun de ses composants internes (Ligne 10: `Interceptor`). Chaque contrôleur intercepteur intercepte les appels entrants et/ou sortants de/vers ses composants sous-jacents et les transmet au contrôleur `Dispatcher` de son parent. En plus de la modification ADL décrite ci-dessus, un

fichier de configuration Julia doit être défini pour inclure la description complète de chaque contrôleur: la classe implémentant son comportement, le type d'intercepteur et une expression de composition des contrôleurs.

```
1 <component name="c">
2     // interface declarations
3     // the content declaration if "c" is a primitive
4     // else for each inner component ci
5     <component name= "ci">
6       // interface declarations of the inner component
7       // the content declaration if "ci" is a primitive
8       // or the inner components declaration if "ci" is a composite
9       // binding declarations if "ci" is a composite
10      <controller desc = "Interceptor"/>
11    </component>
12  <controller desc = "CControllerName"/>
13 </component>
```

Listing A.2: La modification du fichier ADL nécessaire si la vue d'origine correspond à la vue désirée

A.3.2.3 Cas 2. Vue courante ≠ Vue désirée:

Ici nous avons besoin de reconfigurer le système pour qu'il accepte la vue requise. Grâce au composant partagé en Fractal, ce type de reconfiguration est possible. Dans Fractal nous adoptons la stratégie de reconfiguration suivante: un nouveau composant composite est créé comme un composant interne au parent commun le plus proche des composants requis. Ce composite partage les composants requis avec leurs parents d'origine. De façon similaire au cas précédent, le contrôleur composable est intégré à la membrane de ce nouveau composite et un contrôleur intercepteur est ajouté à la membrane de chacun de ses composants internes. Dans ce cas, la vue est ajoutée et la configuration d'origine est préservée. Le choix de la position de la nouvelle déclaration de composite est fait afin de synchroniser le cycle de vie de la vue avec le cycle de vie de ses composants internes. Ainsi, lorsque tous les composants internes de la vue sont détruits, la vue est automatiquement détruite. En outre, lorsque deux vues se recoupent et une interférence entre ces contrôleurs apparaît, les deux vues sont réunies à l'aide de l'opérateur d'union ($\oplus$). Cela permet la composition des contrôleurs composables de vues différentes pour résoudre les interférences éventuels entre eux.

A.3.2.4 Le tisseur Fractal

La reconfiguration des systèmes à composants afin de créer des vues devient une tâche fastidieuse. Notre approche permet d'automatiser cette tâche. Pour cette raison nous avons développé un outil pour Fractal que nous appelons tisseur Fractal (Fractal weaver), conçu une extension de haut niveau de la mise en œuvre de Fractal Julia. La figure A.1 montre l'architecture de l'outil. Dans la figure, l'outil se compose de trois modules: *analyseur VIL*, *transformateur ADL*, et *générateur de configurateurs Julia*. Dans la suite nous décrivons le rôle de chaque module.

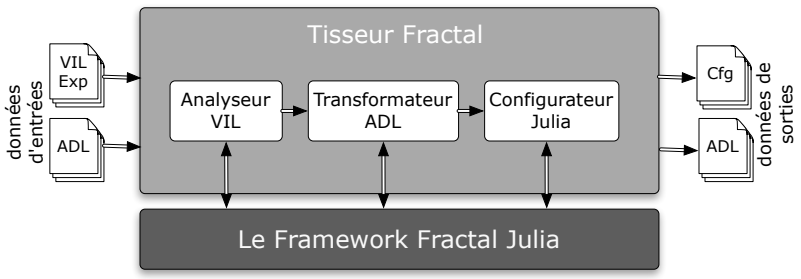

Figure A.1: L'architecture du Tisseur Fractal

Analyseur VIL:

Ce module analyse les expressions VIL en entrée, utilise le mécanisme d'introspection de Julia pour parcourir l'architecture des systèmes et renvoie les points de jointure qui vont être interceptés. Quand une expression n'est pas bien définie, une exception est levée indiquant une erreur dans l'expression d'entrée, sinon un ensemble de points de jointure est renvoyé.

Transformateur ADL:

Ce module utilise les points de jointure retournés à partir du module ci-dessus et les utilise pour déterminer et définir la vue correspondante. Ce transformateur ajoute de nouveaux composites aux systèmes, déclare des contrôleurs composables et des contrôleurs intercepteurs à l'ensemble des composants de la vue, en plus il gère la composition des vues si nécessaire. Le transformateur ADL utilise une structure intermédiaire DAG pour transformer la description ADL d'origine en une autre qui inclut les vues et les contrôleurs nécessaires. Le listing A.3 décrit l'interface implémentée par le transformateur ADL. Il définit trois méthodes: *Adl2Dag()* et *Dag2Adl()* pour transformer une description ADL en une structure de DAG et vice versa, en plus d'une méthode *run()* qui appelle *Adl2Dag()* pour obtenir le DAG de l'architecture d'origine, puis il appelle l'analyseur VIL pour définir la vue nécessaire et mettre à jour la structure du DAG considérant la vue. Finalement, il fait appel à *Dag2Adl()* afin d'obtenir la description ADL finale.

```
1 interface IADLTransformer {
2   Dag Adl2Dag(File src);
3   File Dag2Adl(Dag dag);
4   run(ComponentArchitecture ca, String asId, VExp exp);
5 }
```

Listing A.3: L'interface du Transformateur ADL

Générateur de configurateurs Julia:

Ce dernier module génère des fichiers de configuration Julia nécessaires fournissant des informations sur la mise en œuvre des contrôleurs composables et des inter-

cepteurs. Ces fichiers sont nécessaires pour l'exécution correcte du système sous la plateforme Julia.

A.3.3 VIL en EJB

Pour les modèles à composants plats, tels que EJB [Burke 2006], une vue ne peut être modélisée comme un composite car la hiérarchie des composants n'est pas prise en charge par ces modèles. En outre, les contrôleurs et le partage des composants ne sont plus possibles. Une solution consiste à modéliser un wrapper comme une paire de composants réguliers ou des *beans* `(Dispatcher,Aspect)`. Le `Dispatcher` est lié à tous les composants dont les interfaces doivent être interceptées. Le `Dispatcher` dans ce cas n'a pas une structure générale (aucun nombre ou type d'interfaces prédéfini), mais il peut être généré automatiquement à partir des définitions des points de jointure. Comme celui en Fractal, le `Dispatcher` intercepte tous les appels, définis par des expressions VIL, au lieu de leurs composants originaux, il réifie et transmet ces appels au composant `Aspect`. Le composant `Aspect` exécute un extra-code et décide de poursuivre ou d'ignorer l'appel en retournant *proceed* ou *skip* au `Dispatcher`. L'implémentation de cette solution nécessite la configuration du système en ajoutant les composants `Dispatcher` et `Aspect` et la mise à jour des liaisons pour satisfaire la vue requise comme le montre la figure A.2. Un opérateur de composition peut également être modélisé comme un composant régulier qui intercepte les appels aux points de jointure et les transmet aux composants `Aspects` selon sa stratégie appropriée, puis renvoie le résultat au composant `Dispatcher`.

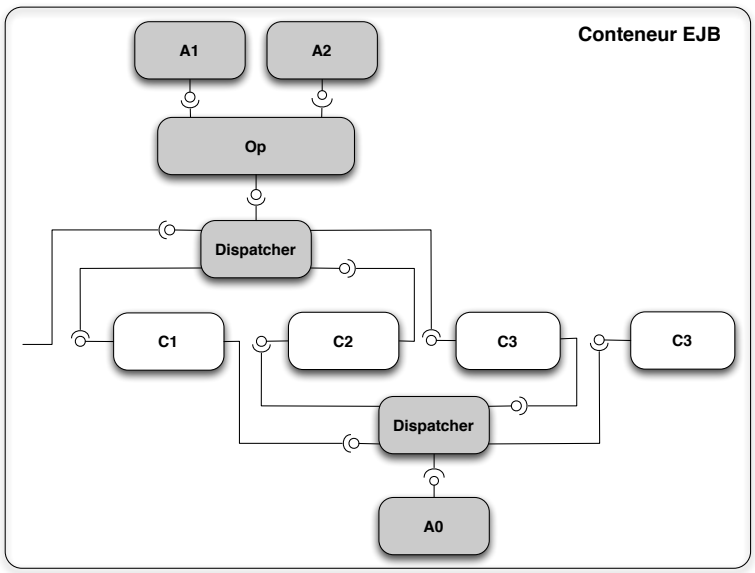

Figure A.2: Les wrappers et leur composition dans modèles à composants plats

A.4 Les interférences des aspects

Dans notre processus d'évaluation, nous nous concentrons sur la détection des interférences entre les aspects et leur résolution. Par la suite nous listons l'ensemble des conclusions déduites lors de l'examination d'un ensemble de modèles à aspects.

1. La vérification syntaxique est une première étape vers la détection d'interférences. De nos jours, il est connu que ces approches ne sont pas assez efficaces pour détecter toutes les interférences possibles entre les aspects, et la vérification sémantique devrait être envisagés.

2. Comme montré dans [Fraine 2008, Weston 2007], une détection des interférences sémantiques efficace doit inclure à la fois l'analyse de flux de contrôle et de données. Cela permet une analyse précise et complète des interactions des aspects.

3. Une spécification aspect devrait être indépendante des autres aspects, sinon une nouvelle spécification est nécessaire à chaque fois que l'aspect nécessite d'être tissé à un nouveau contexte. La spécification de composition d'intentions fournie par Marot et al. [Marot 2009] est un bon exemple de la mise en œuvre de cet effet, où *les intentions* sont définies de manière abstraite et ne comptent pas sur un contexte spécifique.

4. La spécification de la précédence entre les aspects est le moyen fondamental et intuitif pour composer des aspects, malheureusement, cela ne résout pas toutes sortes d'interférences. Ainsi, nous devrions nous débarrasser de la priorité entre les aspects et réfléchir à des stratégies de composition plus puissantes.

5. Les stratégies des compositions doivent être génériques et non pas ad hoc, ce qui permet la réutilisation des solutions proposées.

6. Les approches AOP courantes spécifient d'une manière implicite l'action des aspects sur les points de jointure (proceed or skip). Afin de résoudre les interférences entre les aspects, leurs actions doivent être considérées afin de décider si les advices d'autres aspects sur les points de jointure en question doivent être exécutés ou pas. Dans [Marot 2009], l'action skip est modélisée par l'action *NoProceed* pour résoudre le conflit entre les aspects.

7. Généralement, une interférence ne peut être résolue automatiquement, mais des informations sur l'interférence doivent être rapportées à l'utilisateur, ce qui lui permet de détecter la source du conflit et prendre la bonne décision. C'est le cas pour le modèle basé UTP fourni par Chen et al. [Chen 2010].

Comme conclusion générale pour la résolution des interférences d'aspects, les actions sur les différents points de jointure doivent être explicitement spécifiées dans le comportement des aspects, et des stratégies de composition d'aspects générales et réutilisables doivent être fournies. En outre, le développeur doit être informé

de pourquoi et comment une interférence apparaît. Cela permet au développeur de choisir la bonne stratégie de composition qui résout cette interférence. Nous pensons que les model checkers peuvent aider à la détection des interférences et donner suffisamment d'informations sur la violation des propriétés modélisant le comportement attendu des aspects. Nous croyons également qu'une bibliothèque de patrons de composition peut aider à déterminer la bonne stratégie de composition qui résout une interférence détectée.

A.4.1 Détection et résolution des interférences

Dans cette section, nous montrons comment les interférences entre les aspects peuvent être détectées et potentiellement résolues en utilisant des model checkers et des opérateurs de composition, respectivement. Dans notre proposition nous adoptons l'utilisation du model checker Uppaal [Behrmann 2004] pour la modélisation des systèmes à composants, des aspects, et des systèmes à composants aspectualisés pour la détection formelle des interférences aspect-base et aspect-aspect. Tout d'abord, nous donnons un bref aperçu de Uppaal. Deuxièmement, nous décrivons un système de transformation des systèmes à composant en Uppaal. Troisièmement, nous montrons comment le système de base et les propriétés des aspects peuvent être spécifiées sous forme des formules CTL [Henzinger 1992] (Computational Tree Logic) et vérifiées par Uppaal pour détecter les interférences. Enfin, pour la résolution des interférences nous proposons un ensemble d'opérateurs de composition, chaque opérateur forme un patron de composition qui peut être instancié pour n'importe quels aspects à n'importe quel ensemble des points de jointure.

A.4.1.1 Aperçu de Uppaal

Uppaal [Bengtsson 1996, Larsen 1997, Behrmann 2004] est une boîte à outils pour la conception, simulation et la vérification des propriétés des systèmes qui peuvent être modélisés sous forme des machines d'état étendues avec des variables locales, des types de données, et des variables d'horloge. Ce genre de machines d'état avec le support du temps, sont appelées des automates temporisés [Alur 1992]. Chaque machine d'état en Uppaal est appelée un modèle. Un modèle peut être paramétré avec des constantes et des variables de données indiquant comment ce modèle est instancié (*e.g.,* combien d'instances doivent être créées). Chaque instance est appelée un processus. La boîte à outils Uppaal se compose de trois parties: un langage descriptif de modélisation qui permet la spécification des automates, un simulateur qui permet de visualiser le système en cours d'exécution et le changement des états des variables, et un vérificateur qui permet de vérifier si des propriétés du système sont satisfaites ou pas. Le cas échéant, le vérificateur renvoie une trace d'exécution qui permet de déterminer et de résoudre l'échec.

A.4.1.2 Modélisation des systèmes à composants en Uppaal

Notre proposition consiste à enrichir les ADL(s) actuels des informations nécessaires pour détecter et résoudre les interférences entre les aspects tissés aux systèmes à composants. Cet enrichissement consiste à unifier et explicitement spécifier les comportements des composants primitifs et des aspects sous forme de machines d'états finis, et décrire dans l'ADL un ensemble de règles de tissage pour spécifier où et comment les aspects sont tissés et composés au système de base.

A.4.1.3 Modélisation des composants primitifs

Nous modélisons chaque composant primitif par un processus Uppaal. Cette spécification doit être transformée en formalisme Uppaal-XTA [Bengtsson 1996]. Dans notre ADL, la spécification de comportement est choisie pour être un sous-ensemble de Uppaal-XTA. Dans la spécification, chaque transition est étiquetée par le nom du composant, nom de l'interface et la signature de service séparés par `"_"`. Cette notation permet, par la suite, la modélisation d'assemblage des composants. En outre, lorsque plus d'une instance d'un composant est nécessaire, les noms des transitions sont suffixés avec `"[id]"` indiquant la référence de chaque instance; où `id` est une constante qui varie entre `[1..n]` et `n` est le nombre d'instances indiquées dans la spécification du composant. En Uppaal-XTA, ce nombre devrait être déclaré comme un paramètre du processus modélisant le composant.

A.4.1.4 Modélisation des composants composites

Un composite est modélisé par un ensemble de processus Uppaal, un pour chaque interface liée. Chacun de ces processus a un état central initial et un ensemble de transitions formant des cycles, un pour chaque service. Les services asynchrones sont représentés par des cycles de deux transitions: reçoit un message ($\texttt{cId}_1\texttt{\_itfId}_1\texttt{\_sId?}$), puis le transmet ($\texttt{cId}_2\texttt{\_itfId}_2\texttt{\_sId!}$), tandis que les services synchrones sont représentées par des cycles de quatre transitions: reçoit un message ($\texttt{cId}_1\texttt{\_itfId}_1\texttt{\_sId?}$), le transmet ($\texttt{cId}_2\texttt{\_itfId}_2\texttt{\_sId!}$), attend la réponse ($\texttt{E\_cId}_2\texttt{\_itfId}_2\texttt{\_sId?}$), et transmet la réponse ($\texttt{E\_cId}_1\texttt{\_itfId}_1\texttt{\_sId!}$). Lorsque le composite a plusieurs occurrences, semblables à des primitives, nous suffixons les noms des transitions par `"[id]"`.

A.4.1.5 Modélisation des assemblages des composants

L'assemblage des composants peut être modélisé soit par des processus séparés qui reçoivent un message à partir d'une interface requise et le transmettent à leur interface offerte liée, soit par renommage. Dans notre approche, nous adoptons la seconde solution pour l'optimisation du nombre d'états. Par renommage, une interface $\texttt{itfId}_1$ d'un composant $\texttt{cId}_1$ liée à une interface $\texttt{itfId}_2$ d'un composant $\texttt{cId}_2$ est modélisée en remplaçant chaque nom de transition $\texttt{cId}_1\texttt{\_itfId}_1\texttt{\_sId}$ par $\texttt{cId}_2\texttt{\_itfId}_2\texttt{\_sId}$ dans le processus modélisant le composant $\texttt{cId}_1$, pour chaque nom

de service `sId`. Cela synchronise les transitions des deux composants assemblés aux interfaces liées ($\texttt{itf}_1$ et $\texttt{itf}_2$).

A.4.1.6 Modélisation des systèmes à composants

Un système à composant complet sans aspects est modélisé par la composition parallèle de tous les processus modélisant les composants de l'architecture après la génération des processus modélisant les composites et l'application des règles d'assemblage.

A.4.1.7 Modélisation des aspects

Le comportement de chaque aspect est définit par une machine d'état fini qui se compose d'un ensemble de cycles, de et vers l'état initial, dont chacun décrit le comportement d'un aspect pour un point de jointure donné. La décision de poursuivre ou d'ignorer un appel à un point de jointure appartenant à un point de coupure `pctId` est modélisée par les transitions (`proceed_pctId!`) et (`skip_pctId!`), respectivement. La machine d'état modélisant un aspect est abstraite (il n'y a que des $\texttt{pctId}_i$ utilisés dans les étiquettes des transitions), pour être instanciées, les occurrences `pctId` doivent être remplacées par des points de jointures concrèts. Dans notre modèle, des mapping des points de coupure et leurs descriptions en VIL `(pctId, vexp)` sont donnés dans l'ADL. VIL interprète et transforme les expressions `vexp` de chaque point de coupure en un ensembles de tuples de la forme `(cId, itfId, sId)` (identificateur de composant, identificateur d'interface et signature de service). Pour instancier un aspect, pour chaque point de jointure `(cId, itfId, sId)` dans l'expression correspondante au `pctId` on génère un cycle qui correspond à `pctId` dans la machine d'état abstraite en remplaçant toute occurrence de `pctId` par la concaténation `cId_itfId_sId`.

A.4.1.8 Modélisation du tissage d'aspects

Pour le tissage d'un aspect, un ensemble d'états et de transitions doit être ajouté aux spécifications des composants. Cette extension assure que chaque appel à un point de jointure `jp` (`jp?`) est transmis à l'aspect `aId` (`aId!`) qui exécute son propre comportement et renvoie soit (`proceed_jp!`) pour poursuivre l'appel soit (`skip_jp!`) pour l'ignorer. Dans la machine d'état de composant, quand un aspect reçoit (`proceed_jp?`) il passe à l'état juste avant l'interception de l'appel dans la version d'origine, et quand il reçoit (`skip_jp?`) on distingue deux cas : (1) l'appel est synchrone et dans ce cas il passe à l'état après avoir reçu la fin de l'appel (`E_jp?`), (2) l'appel est asynchrone et dans ce cas le processus passe à l'état initial. En résumé, le processus de tissage d'aspect se fait en deux étapes: (1) l'instanciation de la machine d'état de l'aspect et (2) l'adaptation des composants afin de les synchroniser avec le processus de l'aspect.

A.4.1.9 Modélisation des opérateurs de composition

De façon similaire à un aspect, un opérateur est modélisé par une machine d'état abstraite décrivant où chaque appel intercepté doit être transmis. Pour être instanciées, on remplace les occurrences des "`left`" and "`right`" dans les étiquettes de transitions par les noms des processus aspects. En plus, comme le cas des aspects, des cycles de transitions sont générés pour chaque point de jointure qui correspond à un $\texttt{pctd}_i$ donné dans spécification de l'opérateur.

A.4.1.10 Le processus de détection et de résolution des interférences

Pour la détection des interférences entre les aspects, deux types de propriétés doivent être spécifiées en CTL : (1) les propriétés attendues du système de base, et (2) les propriétés attendues de chaque aspect. Cependant, certains aspects sont conçus pour changer le comportement du système de base, dans ce cas, l'ensemble des propriétés du système qui ne doivent pas être modifiées par chaque aspect doit être explicitement défini par le concepteur. Toutes ces propriétés doivent être passées au model checker pour être vérifiées sur l'automate modélisant le système aspectualisé. Une interférence est détectée si l'une des propriétés des aspects tissés et celles du système de base qui doit être préservée après le tissage des deux aspects, n'est pas vérifiée. Dans ce cas, le model checker renvoie la trace d'exécution où l'interférence apparaît; le concepteur est alors chargé de déterminer la source d'erreur et de trouver la stratégie de composition d'aspects résolvant le problème. Dans notre approche, nous proposons un catalogue des opérateurs de composition sous forme de patrons décrivant la structure abstraite de chaque opérateur (machine d'état finis abstraite). La sémantique de chaque opérateur est définie à l'aide d'un tableau spécifiant la poursuite ou l'ignorance de chaque point de jointure selon son type (partagé ou ne concerne que l'un des aspects) et la décision des aspects opérants (voir le tableau A.3). En plus, des cas d'utilisation et un exemple de motivation de chaque opérateur sont donnés. Dans notre catalogue, nous proposons six opérateurs, mais le catalogue peut être étendu avec d'autres opérateurs de la même façon. Notre catalogue n'est pas conçu pour être complet mais il forme une première étape vers une étude plus exhaustive des interactions des aspects.

LHS	RHS	Seq(LHS,RHS)
Shared join points		
proceed	proceed	proceed
skip	proceed	proceed
proceed	skip	proceed
skip	skip	skip
LHS join points		
proceed	-	proceed
skip	-	skip
RHS join points		
-	proceed	proceed
-	skip	skip

Table A.3: La sémantique de l'opérateur `Seq`

En résumé, cette partie décrit notre contribution dans la détection et la résolution des interférences d'aspects dans les systèmes à composants. L'approche propose un ADL pour la spécification des structures et des comportements des aspects et des composants et leurs vérification formelle à l'aide de model checkers comme Uppaal. Un ensemble de règles de transformation de l'ADL vers des machines d'état est présenté. Nous avons montré comment on peut détecter les interférences entre les aspects en spécifiant au model checker l'ensemble des propriétés à vérifier. En outre, un ensemble d'opérateurs de composition est utilisé pour résoudre les interférences. Nous devons mentionner que notre approche ne repose ni sur un modèle à composant spécifique, ni sur un model checker particulier. Les règles de transformation sont générales, mais non exhaustives car d'autres modèles à composants peuvent exiger d'autres règles pour transformer leurs propres éléments d'architecture logicielle. Par exemple, SOFA 2 [Hnetynka 2007, Bures 2006] propose des connecteurs pour l'assemblage des composants, ces connecteurs sont définis sous forme d'entités de première classe qui mettent en œuvre différentes stratégies de communication. Ces connecteurs peuvent être modélisés comme des processus Uppaal qui modélisent les stratégies de communication désirées. Par ailleurs, l'ensemble des opérateurs de composition proposé est un ensemble extensible et d'autres opérateurs peuvent être définis de manière similaire.

A.5 Conclusion générale

La programmation par aspects (AOP) fournit un mécanisme pour faire face aux fonctionnalités non modulaires des systèmes logiciels en assurant leurs modularités et leurs réutilisabilités. La programmation par composants (CBSE) fournit un mécanisme modulaire pour la conception des systèmes logiciels en améliorant leurs réutilisabilités et en réduisant l'effort de maintenance. L'idée derrière AOP est de modéliser une propriété non modulaire par une entité séparée nommée aspect. Un aspect consiste en la définition d'une ou de plusieurs méthodes, appelées advices, décrivant le comportement de la propriété et l'utilisation d'un langage expressif pour décrire des points dans le programme de base, nommés points de jointure, qui précisent où les advices doivent être exécutés. En outre, l'AOP fournit un mécanisme qui permet le tissage du code des advices dans les bons endroits dans le programme de base. D'autre part, le CBSE divise un système logiciel en plusieurs entités, nommés composants, et fournit un mécanisme pour les composer afin d'obtenir le système requis complet. Cependant, comme les autres paradigmes, le CBSE ne peut modéliser les fonctionnalités non modulaires, qui sont dispersées sur plusieurs composants, comme des entités régulières. Afin d'assurer une meilleure modularité et réutilisabilité des composants logiciels en CBSE, les propriétés non modulaires doivent être ajoutées de façon modulaire et par conséquent les aspects devraient être intégrés dans les modèles à composants. Pour atteindre cet objectif, une intégration des aspects et de leurs constituants dans les modèles à composants est nécessaire. Dans cette thèse, nous avons proposé une approche générique d'intégration des as-

pects dans les modèles à composants. L'approche proposée bénéficie de la diversité des configurations des systèmes à composants pour spécifier la configuration adéquate pour chaque aspect. Toutefois, l'ajout de plusieurs aspects peut donner lieu à des interférences lorsque l'exécution d'un aspect peut modifier l'exécution d'un autre. Pour cette raison, nous avons fourni un support formel pour la détection et la résolution des interférences des aspects dans les modèles à composants.

Dans notre proposition, les aspects sont modélisés par des enveloppes (wrappers) sur des vues. Une vue est une configuration adéquate pour un wrapper, elle encapsule tous les composants d'un wrapper dans le même composite. Les vues sont clairement visibles dans les modèles hiérarchiques tels que Fractal, où les composites sont créés pour encapsuler les composants d'un wrapper. Dans les modèles à composants plats tels que EJB, une vue est abstraite, elle ne définit que les points où un wrapper doit être ajouté, un wrapper peut se modéliser sous forme d'un ou de plusieurs composants ou connecteurs réguliers liés aux autres composants de la vue. Pour la définition déclarative des vues, nous avons introduit VIL : un langage des points de jointures adopté pour les modèles à composants. Pour la détection des interférences et leurs résolution, un processus de formalisation est défini : tout d'abord, un langage de description d'architectures (ADL) abstrait est proposé pour décrire la structure ainsi que les comportements des composants. Les composants primitifs, les aspects et leurs comportements sont donnés dans ADL sous forme de machines d'état finis avec une notation spécifique pour les transitions. Ensuite, un système de transformation est exécuté pour générer la formalisation complète du système aspectualisé (le système de base avec des aspects tissés). Le processus de transformation inclut l'adaptation des processus des composants primitifs, la génération des processus modélisant les composants composites, et le tissage des aspects. Le système résultant peut être utilisé comme entrée au modèle de vérification formelle avec l'ensemble des propriétés intrinsèques du système et les propriétés attendues de chaque aspect. Une interférence est détectée lorsque au moins une des propriétés attendues d'un aspect ou une des propriétés intrinsèques du système est violée. Pour résoudre les interférences, un catalogue des patrons des opérateurs de composition est fourni. Pour la vérification formelle nous avons utilisé le vérificateur des modèles Uppaal. L'approche proposée est conçue pour être générale (elle ne repose pas sur un modèle de composant spécifique), nous avons montré comment les vues peuvent être modélisés et comment les wrappers peuvent être mis en œuvre pour deux catégories différentes de modèles à composants: les modèles hiérarchiques avec partage de composants tels que Fractal et les modèles plats comme EJB.

A.5.1 Les modèles à composants aspectualisés

Il y a plusieurs façons d'étendre les modèles à composants avec les concepts d'AOP, chaque méthode dépend des fondements de chaque modèle. Cependant, un effort est nécessaire pour étendre à chaque modèle. Une approche unifiée est alors indispensable pour atténuer ces efforts. Le concept de vues proposé dans cette thèse est générique et peut être adapté à différents modèles à composants. Par ailleurs, les

stratégies classiques de tissage (complile-time, runtime) détruisent la modularité des composants et compliquent fortement le tissage/détissage dynamique des aspects. Pour pallier ce problème, le wrapping ou le monitorat des composants est le mécanisme efficace pour les modèles à composants. Cela préserve la modularité à la fois des composants et des aspects avant et après le tissage et permet le tissage/détissage dynamique des aspects en activant/désactivant les wrappers. Les vues permettent de localiser le champ d'application de chaque aspect et peuvent être vues clairement dans les modèles à composants hiérarchiques, par contre, dans les modèles plats les vues sont des unités abstraites où seuls les points d'intervention d'un aspect sont spécifiés et des composants supplémentaires et/ou des connecteurs sont utilisés pour transférer les appels interceptés aux composants modélisant les aspects. L'ajout des composants et des connecteurs supplémentaires durcit et complique l'architecture originale des systèmes. Notez que les vues peuvent être définies pendant différentes phases du processus de développement des logiciels. Par exemple, dans la phase de conception, les vues peuvent être définies pour déterminer l'architecture adéquate aux propriétés non fonctionnelles qui sont déjà connues. En outre, les vues peuvent être utilisées dans la phase de maintenance où de nouvelles exigences non fonctionnelles sont requises pour un système déjà conçu et mis en œuvre.

A.5.2 Analyse d'interaction des aspects

Lorsque plusieurs aspects sont tissés au système, leurs interactions doivent être soigneusement analysées afin d'éviter les interférences possibles qui peuvent conduire à un mauvais comportement du système complet. Cependant, les interférences ne sont pas toujours évitables par des opérateurs de composition, cela dépend de la nature de l'interaction et de l'effet de bord des aspects sur le système. Les aspects interagissent de différentes façons, notre expérience montre que trois types d'interactions fondamentaux des aspects sont disponibles:

Neutre: une interaction entre deux aspects est neutre quand les aspects sont tissés au même système, le système résultant satisfait les propriétés attendues des deux aspects tissés en plus de l'invariant du système.

Interactions positives: deux aspects interagissent positivement lorsque l'exécution d'un aspect renforce l'exécution de l'autre en satisfaisant la condition du second aspect.

Interactions négatives: cette interaction négative apparaît lorsque le tissage d'un aspect individuel à un système est correct, mais son tissage à un système avec des aspects déjà tissé conduit à des comportements indésirables. Cela signifie que l'exécution d'un aspect influe sur l'exécution de l'autre.

L'analyse des interférences des aspects est un travail fastidieux où les interactions d'aspects doivent être soigneusement analysés. Ceci peut être réalisé en spécifiant clairement ce qui est attendu par un système avant et après le tissage et les propriétés

attendues de chaque aspect. Ceci permet la détection de propriétés contradictoires, qui est une source d'erreurs si elle est faite manuellement. Des outils tels que les model checkers sont utiles pour détecter de telles propriétés contradictoires. En raison du problème d'explosion combinatoire d'état, l'utilisation de ces outils est encore limitée aux systèmes ayant une taille faible en nombre des composants et/ou dotés de spécifications de comportements optimisées. En outre, notre proposition d'un catalogue d'opérateurs de composition permet aux utilisateurs de choisir la stratégie de composition adéquate qui résout les interférences détectées. Une solution alternative est l'adaptation des aspects où chaque aspect devrait être conscient de la présence ou l'absence d'autres aspects afin qu'il puisse décider que faire avec chaque point de jointure intercepté. Cependant, l'adaptation des aspects est difficile surtout dans le cas où les aspects contradictoires ne peuvent être facilement et statiquement déterminés. Mais lorsque les opérateurs ne fournissent pas une solution, l'adaptation peut être utilisée.

A.5.3 Perspectives

Outre la généralité de notre approche, certaines limites sont à relever, nous proposons les solutions suivantes: (1) l'utilisation des model checkers pour les systèmes à composants de taille énorme, suivant notre approche, conduit à une explosion combinatoire d'états. Pour remédier à ce problème, nous avons adopté l'abstraction des systèmes où les composites, avec leurs composants internes qui ne sont pas concernés par un aspect, sont remplacés par une machine d'état primitive simplifiée modélisant le comportement externe de tels composites. Cependant, cette tâche est difficile et peut masquer des détails de comportements important pour la détection des interférences. Une solution consiste à utiliser les mÃ©thodes de preuves tels que la méthode B. En utilisant B, chaque composant peut être modélisé par une machine abstraite qui spécifie le comportement de chaque composant en termes de pré/post conditions pour chaque service et un protocole qui décrit le flux de contrôle de chaque composant. Ainsi, l'outil Atelier B pourrait être utilisé pour vérifier la conformité des propriétés intrinsèques des systèmes avec celles des différents aspects pour détecter les interférences. (2) le catalogue des opérateurs de composition doit être étendu. Cela peut être atteint par l'expérimentation de l'approche avec plus d'exemples concrets et d'autres interactions d'aspects. (3) vu que les vues sont définies pour les systèmes centralisés, où les aspects sont définis pour des composants dans le même serveur, dans les systèmes distribués, les vues doivent être adaptées et les aspects distribuées avec leurs propriétés doivent être pris en compte.

Bibliography

[Adamek 2007] Jiri Adamek, Tomas Bures, Pavel Jezek, Jan Kofron, Vladimir Mencl, Pavel Parizek and Frantisek Plasil. Component reliability extensions for fractal component model. Academy of Sciences of the Czech Republic and France Telecom. http://websvn.ow2.org/, 2007. (Cited on pages 12 and 138.)

[Ahmed 1996] Mohsin Ahmed and G. Venkatesh. *Dense Time Logic Programming.* Journal of Symbolic Computation, vol. 22, no. 5-6, pages 585 – 613, 1996. (Cited on page 114.)

[Aksit 1992] Mehmet Aksit, Lodewijk Bergmans and Sinan Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach.* In ECOOP '92 European Conference on Object-Oriented Programming, volume 615 of *LNCS*, pages 372–395, Berlin, Germany, 1992. Springer. (Cited on page 22.)

[Allan 2005] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam and Julian Tibble. *Adding Trace Matching with Free Variables to AspectJ.* ACM SIGPLAN Notices, vol. 40, no. 10, pages 345–364, 2005. (Cited on pages 21 and 57.)

[Alur 1992] Rajeev Alur and David Dill. *The theory of timed automata.* In J. de Bakker, C. Huizing, W. de Roever and G. Rozenberg, editeurs, Real-Time: Theory in Practice, volume 600 of *Lecture Notes in Computer Science*, pages 45–73. Springer Berlin / Heidelberg, 1992. (Cited on pages 114 and 177.)

[Avgustinov 2006] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie Hendren, Ondřej Lhoták, Oege de Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble and Mathieu Verbaere. *Aspects for Trace Monitoring.* In Formal Approaches to Software Testing and Runtime Verification, volume 4262 of *LNCS*, pages 20–39. Springer Berlin / Heidelberg, 2006. (Cited on page 57.)

[Behrmann 2004] Gerd Behrmann, Alexandre David and Kim G. Larsen. *A Tutorial on Uppaal.* In SFM-RT: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, numéro 3185 de LNCS, pages 200–236. Springer-Verlag, 2004. (Cited on pages 114 and 177.)

[Ben-Ari 2008] Mordechai Ben-Ari. Principles of the spin model checker. Springer, 1st édition, 2008. (Cited on page 34.)

[Bengtsson 1996] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. *UPPAAL-a tool suite for automatic verification of real-time systems*. In Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control, pages 232–243. Springer-Verlag, 1996. (Cited on pages 114, 177 and 178.)

[Bergmans 1996] Lodewijk Bergmans and Mehmet Akşit. *Composing synchronization and real-time constraints*. Journal of Parallel and Distributed Computing, vol. 36, no. 1, pages 32–52, July 1996. (Cited on page 12.)

[Bergmans 2001] Lodewijk Bergmans and Mehmet Aksit. *Composing crosscutting concerns using composition filters*. Communications of the ACM, vol. 44, pages 51–57, 2001. (Cited on page 23.)

[Bruneton 2004] E. Bruneton, T. Coupaye and J. B. Stefani. *The Fractal Component Model*. Online Available: http://fractal.ow2.org/specification/index.html, 2004. (Cited on pages 12, 53, 96, 101, 118 and 171.)

[Bruneton 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma and Jean-Bernard Stefani. *The Fractal Component Model and its Support in Java*. Software-Practice and Experience, vol. 36, no. 11-12, pages 1257–1284, 2006. (Cited on page 55.)

[Bures 2006] Tomas Bures, Petr Hnetynka and Frantisek Plasil. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on pages 135 and 181.)

[Burke 2006] Bill Burke and Richard Monson-Haefel. Enterprise javabeans 3.0 (5th edition). O'Reilly Media, Inc., 2006. (Cited on pages 41, 168 and 175.)

[Chen 2008] Zhenbang Chen, Abdel Hakim Hannousse, Dang Van Hung, Istvan Knoll, Xiaoshan Li, Zhiming Liu, Yang Liu, Qu Nan, Joseph C. Okika, Anders P. Ravn, Volker Stolz, Lu Yang and Naijun Zhan. *Modelling with Relational Calculus of Object and Component Systems - rCOS*. In The Common Component Modeling Example, volume 5153 of *Lecture Notes in Computer Science*, pages 116–145. Springer Berlin / Heidelberg, August 2008. (Cited on page 164.)

[Chen 2010] Xin Chen, Nan Ye and Wenxu Ding. *A formal approach to analyzing interference problems in aspect-oriented designs*. In Proceedings of the Third international conference on Unifying theories of programming, UTP'10, pages 157–171. Springer-Verlag, 2010. (Cited on pages 34, 36, 37 and 176.)

[Choi 2000] Jung Pil Choi. *Aspect-Oriented Programming with Enterprise JavaBeans*. In Proceedings of the 4th International conference on Enterprise

Distributed Object Computing, EDOC '00, pages 252–261, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on pages 42, 44 and 167.)

[Clemente 2002] Pedro J. Clemente, Juan Hernández, Juan M. Murillo, Miguel A. Pérez and Fernando Sánchez. *AspectCCM: An Aspect-Oriented Extension of the Corba Component Model*. In Proceedings of 28th EUROMICRO conference, pages 10–16, Los Alamitos, CA, USA, 2002. IEEE Computer Society. (Cited on pages 46 and 167.)

[Coady 2001] Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn. *Using aspectC to improve the modularity of path-specific customization in operating system code*. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-9, pages 88–98, New York, NY, USA, 2001. ACM. (Cited on page 11.)

[Coupaye 2006] Thierry Coupaye, Vivien Quéma, Lionel Seinturier and Jean-Bernard Stefani. *Le système de composants Fractal*. In ICAR, editeur, Intergiciel et Construction d'Applications Réparties. ICAR, 2006. (Cited on page 53.)

[Dantas 2008] Daniel S. Dantas, David Walker, Geoffrey Washburn and Stephanie Weirich. *AspectML: A polymorphic aspect-oriented functional programming language*. ACM Trans. Program. Lang. Syst., vol. 30, pages 14:1–14:60, May 2008. (Cited on page 11.)

[David 2003] Pierre Charles David and Thomas Ledoux. *Towards a Framework for Self-adaptive Component-Based Applications*. In Distributed Applications and Interoperable Systems, volume 2893 of *LNCS*, pages 1–14. Springer Berlin / Heidelberg, 2003. (Cited on page 58.)

[David 2009a] Pierre-Charles David, Thomas Ledoux, Marc Léger and Thierry Coupaye. *FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures*. Annales des Télécommunications, vol. 64, no. 1-2, pages 45–63, 2009. (Cited on page 55.)

[David 2009b] Pierre-Charles David, Thomas Ledoux, Marc Léger and Thierry Coupaye. *FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures*. Annales des Télécommunications, vol. 64, no. 1-2, pages 45–63, March 2009. (Cited on page 98.)

[de Roo 2008] A. de Roo, M.F.H. Hendriks, W. Havinga, P. Durr and L. Bergmans. *Compose*: a Language- and Platform-Independent Aspect Compiler for Composition Filters*. In First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, 2008. (Cited on page 28.)

[Douence 2002] Rémi Douence, Pascal Fradet and Mario Südhot. *A Framework for the Detection and the Resolution of Aspect Interaction.* In GPCE'06: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative programming and component engineering, pages 173–188. Springer-Verlag, 2002. (Cited on pages 30 and 37.)

[Douence 2004] Rémi Douence, Pascal Fradet and Mario Südhot. *Composition, Reuse and Interaction Analysis of Stateful Aspects.* In ACM Press, editeur, AOSD'04: Proceedings of the 3rd international conference on Aspect-oriented software development, pages 141–150. ACM, 2004. (Cited on page 30.)

[Douence 2006] Rémi Douence, Didier Le Botlan, Jacques Noyé and Mario Südhot. *Concurent Aspects.* In proceedings of the 5th international conference on Generative programming and component engineering, GPCE'06, pages 79–88. ACM, 2006. (Cited on pages 21 and 30.)

[Durr 2007] Pascal Durr, Lodewijk Bergmans and Mehmet Aksit. *Static and dynamic detection of behavioral conflicts between aspects.* In Proceedings of the 7th international conference on Runtime verification, RV'07, pages 38–50, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on pages 27 and 29.)

[Durr 2008] P. E. A. Durr. *Resource-based Verification for Robust Composition of Aspects.* PhD thesis, University of Twente, Enschede, June 2008. (Cited on pages 27, 29 and 37.)

[Fakih 2004] Houssam Fakih, Noury Bouraqadi and Laurence Duchien. *Aspects and Software Components: A case study of the FRACTAL Component Model.* In International Workshop on Aspect-Oriented Software Development (WAOSD 2004), Beijing, China, 2004. (Cited on page 56.)

[Fraine 2008] Bruno De Fraine, Pablo Daniel Quiroga and Viviane Jonckers. *Management of Aspect Interactions using Statically-Verified Control-Flow Relations.* In Proceedings of the 3rd International Workshop on Aspects, Dependencies and Interactions, ADI'08, pages 5–14, 2008. (Cited on pages 33, 35, 37 and 176.)

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides. Design patterns : elements of reusable object-oriented software. Addison-Wesley, first édition, 1995. (Cited on pages 12 and 129.)

[Goldman 2010] Max Goldman, Emilia Katz and Shmuel Katz. *MAVEN: modular aspect verification and interference analysis.* Formal Methods System Designs, vol. 37, pages 61–92, 2010. (Cited on page 31.)

[Hatcliff 2001] John Hatcliff and Matthew B. Dwyer. *Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software.* In Proceedings of the

12th International Conference on Concurrency Theory, CONCUR '01, pages 39–58. Springer-Verlag, 2001. (Cited on page 34.)

[Havelund 2000] Klaus Havelund and Thomas Pressburger. *Model checking JAVA programs using JAVA PathFinder*. International Journal on Software Tools for Technology Transfer (STTT), vol. 2, pages 366–381, 2000. (Cited on page 34.)

[Henzinger 1992] T.A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine. *Symbolic model checking for real-time systems*. Information and Computation, vol. 111, pages 394 –406, 1992. (Cited on page 177.)

[Hnetynka 2007] Petr Hnetynka and Tomás Bures. *Advanced Features of Hierarchical Component Models*. In Proceedings of the 10th International Conference on Information System Implementation and Modeling (ISIM'07), volume 252 of *CEUR Workshop Proceedings*, pages 3–10, Czech Republic, April 2007. CEUR-WS.org. (Cited on pages 135 and 181.)

[Hoare 1998] C.A.R. Hoare and He Jifeng. Unified theories of programming. Prentice Hall, 1998. (Cited on page 34.)

[Jifeng 2006] He Jifeng, Xiaoshan Li and Zhiming Liu. *rCOS: A refinement calculus of object systems*. Theoretical Computer Science, vol. 365, no. 1-2, pages 109–142, July 2006. (Cited on page 164.)

[Katz 2004] Shmuel Katz and Marcelo Sihman. *Aspect Validation Using Model Checking*. In Verification: Theory and Practice, volume 2772 of *LNCS*, pages 391–392. Springer Berlin / Heidelberg, 2004. (Cited on pages 34 and 37.)

[Katz 2008] Emilia Katz and Shmuel Katz. *Incremental analysis of interference among aspects*. In Proceedings of the 7th workshop on Foundations of aspect-oriented languages, FOAL '08, pages 29–38, New York, NY, USA, 2008. ACM. (Cited on pages 31 and 37.)

[Kiczales 2001a] Gregor Kiczales and Erik Hilsdale. *Aspect-oriented programming*. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-9, page 313. ACM, 2001. (Cited on pages 11, 19 and 96.)

[Kiczales 2001b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. *An Overview of AspectJ*. In Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. (Cited on pages 11, 21, 97 and 165.)

[Kienzle 2009] Jörg Kienzle, Ekwa Duala-Ekoko and Samuel Gélineau. *Transactions on Aspect-Oriented Software Development V*. chapitre AspectOptima:

A Case Study on Aspect Dependencies and Interactions, pages 187–234. Springer-Verlag, Berlin, Heidelberg, 2009. (Cited on page 135.)

[Kniesel 2009] Günter Kniesel. *Transactions on Aspect-Oriented Software Development V.* chapitre Detection and Resolution of Weaving Interactions, pages 135–186. Springer-Verlag, Berlin, Heidelberg, 2009. (Cited on pages 31 and 37.)

[Krishnamurthi 2007] Shriram Krishnamurthi and Kathi Fisler. *Foundations of Incremental Aspect Model-Checking.* ACM Transactions on Software Engineering and Methodology, vol. 16, no. 2, pages 1–39, 2007. (Cited on page 31.)

[Larsen 1997] Kim G. Larsen, Paul Pettersson and Wang Yi. *Uppaal in a nutshell.* International Journal on Software Tools for Technology Transfer, vol. 1, pages 134–152, 1997. (Cited on pages 114, 119 and 177.)

[Marot 2009] Antoine Marot and Roel Wuyts. *Detecting unanticipated aspect interferences at runtime with compositional intentions.* In Proceedings of the Workshop on AOP and Meta-Data for Software Evolution, RAM-SE '09, pages 31–35. ACM, 2009. (Cited on pages 28, 35, 36, 37 and 176.)

[Meyer 1997] Bertrand Meyer. Object-oriented software construction. Prentice-Hall, Inc., 2nd édition, 1997. (Cited on page 19.)

[Navasa 2009] Amparo Navasa, Miguel A. Pérez-Toledano and Juan M. Murillo. *An ADL dealing with aspects at software architecture stage.* Information and Software Technology, vol. 51, no. 2, pages 306–324, 2009. (Cited on pages 61, 166, 167 and 168.)

[OMG 2006] OMG. *Corba Component Model Specification 4.0.* Online: http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf, 2006. (Cited on pages 44 and 168.)

[Ossher 2001] Harold Ossher and Peri Tarr. *Using multidimensional separation of concerns to (re)shape evolving software.* Communications of the ACM, vol. 44, pages 43–50, 2001. (Cited on pages 25 and 59.)

[Pawlak 2005] Renaud Pawlak, Lionel Seinturier and Jean-Philippe Retaillé. Foundations of aop for j2ee development. Apress, Berkely, CA, USA, 2005. (Cited on pages 47, 48, 167 and 168.)

[Pérez 2006] Jennifer Pérez, Nour Ali, Jose Carsí and Isidro Ramos. *Designing Software Architectures with an Aspect-Oriented Architecture Description Language.* In CBSE: Component-Based Software Engineering, volume 4063 of *LNCS*, pages 123–138. Springer Berlin / Heidelberg, 2006. (Cited on pages 59 and 168.)

[Pérez 2008] Jennifer Pérez, Nour Ali, Jose A. Carsí, Isidro Ramos, Bárbara Álvarez, Pedro Sanchez and Juan A. Pastor. *Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems*. Information and Software Technology, vol. 50, no. 9-10, pages 969–990, 2008. (Cited on pages 59 and 168.)

[Pessemier 2006] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye and Laurence Duchien. *A Model for Developing Component-Based and Aspect-Oriented Systems*. In Software Composition : 5th International Symposium, SC 2006, volume 4089 of *LNCS*, pages 259–274. Springer Berlin / Heidelberg, 2006. (Cited on pages 57 and 166.)

[Pessemier 2007] Nicolas Pessemier. *Unification des approches par aspects et à composants*. PhD thesis, Université Lille 1, Laboratoire d'Informatique Fondamentale de Lille, Lille, France, 2007. (Cited on page 58.)

[Pessemier 2008] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien and Thierry Coupaye. *A component-based and aspect-oriented model for software evolution*. Int. J. Computer Applications in Technology, vol. 31, no. 1/2, pages 94–105, 2008. (Cited on pages 57 and 166.)

[Pinto 2003] Mónica Pinto, Lidia Fuentes and Jose María Troya. *DAOP-ADL: an architecture description language for dynamic component and aspect-based development*. In Proceedings of the 2nd international conference on Generative programming and component engineering, GPCE '03, pages 118–137. Springer-Verlag, 2003. (Cited on pages 52 and 167.)

[Pinto 2005] Mónica Pinto, Lidia Fuentes and José María Troya. *A Dynamic Component and Aspect-Oriented Platform*. The Computer Journal, vol. 48, no. 4, pages 401–420, 2005. (Cited on pages 52 and 167.)

[Pyarali 1998] Irfan Pyarali and Douglas C. Schmidt. *An overview of the CORBA portable object adapter*. ACM StandardView, vol. 6, no. 1, pages 30–43, 1998. (Cited on page 45.)

[Suvée 2003] Davy Suvée, Wim Vanderperren and Viviane Jonckers. *JAsCo: an aspect-oriented approach tailored for component based software development*. In Proceedings of the 2nd international conference on Aspect-oriented software development, AOSD '03, pages 21–29. ACM, 2003. (Cited on pages 11, 49 and 167.)

[Suvée 2006] Davy Suvée, Bruno De Fraine and Wim Vanderperren. *A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development*. In CBSE: Component-Based Software Engineering, volume 4063 of *Lecture Notes in Computer Science*, pages 114–122. Springer Berlin / Heidelberg, June 2006. (Cited on page 11.)

[Szyperski 2002] Clemens Szyperski, Dominik Gruntz and Stephan Murer. Component software: Beyond object-oriented programming. Component Software Series. ACM Press and Addison-Wesley, New York, NY, 2nd édition, 2002. (Cited on pages 11 and 96.)

[Tarr 1999] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton Jr. *N degrees of separation: multi-dimensional separation of concerns.* In Proceedings of the 21st international conference on Software engineering, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM. (Cited on pages 25 and 59.)

[Tarr 2000] Peri Tarr and Harold Ossher. *Hyper/J user and installation manual.* Rapport technique, IBM T. J. Watson Research Center, 2000. (Cited on page 25.)

[Šery 2007] Ondřej Šery and František Plášil. *Slicing of component behavior specification with respect to their composition.* In Proceedings of the 10th international conference on Component-based software engineering (CBSE'07), volume 4608 of *Lecture Notes in Computer Science*, pages 189–202. Springer-Verlag, 2007. (Cited on pages 12 and 138.)

[Walls 2007] Craig Walls and Ryan Breidenbach. Spring in action. Manning Publications Co., Greenwich, CT, USA, 2007. (Cited on page 47.)

[Wang 2001] Nanbor Wang, Douglas C. Schmidt and Carlos O'Ryan. Overview of the corba component model, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (Cited on page 44.)

[Weston 2007] Nathan Weston, Francois Taiani and Awais Rashid. *Interaction Analysis for Fault-Tolerance in Aspect-Oriented Programming.* In Proceedings of the Workshop on Methods, Models and Tools for Fault-Tolerance, MeMoT'07, pages 95–102, 2007. (Cited on pages 33, 35, 37 and 176.)

Abdelhakim Hannousse

Aspectualizing Component Models: Implementation and Interference Analysis

Résumé

L'utilisation d'AOP pour modéliser les préoccupations transverses ou non modulaire de CBSE assure une meilleure modularité et réutilisabilité des composants. Dans ce cadre, nous proposons une approche générique pour modéliser les aspects dans les modèles à composants. Nous modélisons un aspect par un wrapper sur une vue de système. Une vue décrit une configuration adéquate du système où tous les composants dans l'intérêt d'un aspect sont encapsulés dans le même composite. Pour la définition des vues, nous définissons un langage déclaratif VIL. Nous illustrons comment les vues sont mises en œuvre dans des modèles à composants (ex., Fractal), et nous fournissons un modèle formel pour l'analyse des interférences d'aspects. Les composants et les aspects sont modélisés par des automates et Uppaal est utilisé pour détecter les interférences. Pour la résolution d'interférences, nous fournissons un ensemble d'opérateurs de composition. Notre approche est illustrée par un exemple: l'accès wifi dans un aéroport.

Mots clés

Programmation par aspect, programmation par composants, analyse formelle, composition des aspects

Abstract

Using AOP to model non-modular concerns in CBSE ensures better modularity and reusability of components. In this thesis, we provide a model independent approach for modeling aspects in component models. In the approach we model aspects as wrappers on views of component systems. A view describes an adequate component system configuration where all the components of interest of an aspect are encapsulated in the same composite. For declarative definition of views, we provide a declarative language VIL. We illustrate how views are implemented in component models (e.g., Fractal). We provide a formal framework for aspect interferences analysis. In the framework component systems and aspects are modeled as automata and Uppaal model checker is used for the detection of aspect interferences. For interferences resolution, we provide a set of composition operators as templates to be instantiated for any two arbitrary aspects. Our approach is illustrated with an airport wireless access example.

Keywords

AOP, CBSE, aspect interferences, interferences detection, aspect composition.

l'université nantes angers le mans

PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

ECOLE DES MINES DE NANTES