Badji Mokhtar-Annaba University

Department of Computer Science

Ph.D. Thesis

Discipline: Computer Science

Presented by

Mehdi Hariati

Formal Approach for Combining Aspects and Software Components

---

Supervised by Nadir Farah

Presented on January 31, 2020 before the jury composed of :

Prof. Mohamed Tahar Kimour	University of Annaba	President
Prof. Nadir Farah	University of Annaba	Supervisor
Prof. Allaoua Chaoui	University of Constantine	Examiner
Prof. Makhlouf Derdour	University of Oum El Bouaghi	Examiner

Badji Mokhtar-Annaba University
P.B.12, Annaba, 23000 Algeria.

Cette thèse est dédiée à ma famille.

Le premier acte de la connaissance, c'est sans doute la prise de conscience, l'évaluation, de ce que l'on ne connaît pas !

Remerciements

Je m'empresse tout d'abord à rendre un grand hommage à feu au Professeur Meslati qui nous a malheureusement quitté pour un monde meilleur, sans manquer de le remercier très chaleureusement à titre posthume pour son encadrement et son soutien qui ne s'est jamais démenti, permettant à cette thèse de devenir réalité. Il a fallu bénéficier tout au long de la réalisation de ce travail, de son immense expérience et de ses utiles conseils, cherchant toujours plus à développer mon goût pour les méthodes rigoureuses, qualités essentielles, sans lesquelles, on ne peut vraiment jamais rien concevoir de valable dans ce domaine, si difficile. En me laissant toujours une suffisante liberté de manoeuvre, ses orientations prodiguées à bon escient et au bon moment ont permis de poursuivre et de mener à bien cette démarche dans de très bonnes conditions, qu'il trouve ici, l'expression de ma profonde reconnaissance.

Je remercie également le Professeur Farah qui a bien voulu prendre avec infiniment de bienveillance la relève pour assurer la direction de la présente thèse.

Reste par ailleurs, que ce travail n'aurait pu aboutir sans la riche et fructueuse collaboration que j'ai pu avoir avec le Professeur Poizat dont l'aide précieuse m'a été d'un soutien infaillible, qu'il trouve ici l'expression de ma profonde gratitude ainsi que mes remerciements les plus sincères via ces quelques lignes.

Je remercie le Professeur Lounis pour ses judicieux conseils, faisant qu'à chaque fois il ne cessa de me donner la possibilité de voir le travail entrepris sous un angle nouveau.

J'adresse également mes plus vifs remerciements aux membres du jury : Professeur Chaoui, Professeur Kimour, et Professeur Derdour pour leur infinie patience mise dans l'attentive lecture de ce document ainsi que pour leurs précieuses critiques au combien nécessaire, pour avancer à pas résolus sur le chemin très escarpé du progrès scientifique et technique. Je remercie de façon particulière le Professeur Kimour d'avoir bien voulu prendre sur ses épaules toute la responsabilité qui sied à un président du jury.

Enfin, je remercie pour leur soutient inconditionnel, ma famille et mes amis.

Abstract

Résumé

Dans le présent travail, nous partons du principe qu'une synergie entre le développement basé composant et le développement orienté aspect est une issue prometteuse. En fait, chacun des deux paradigmes peut apporter des avantages pour l'autre et combler ses insuffisances. D'un côté, les aspects seront dotés des mêmes propriétés structurantes que les composants logiciels, permettant ainsi leur réutilisation. D'un autre côté, la puissance expressive de la programmation orientée aspect présente des avantages indéniables, notamment pour l'évolution des composants logiciels, permettant de ce fait, leurs adaptation aux changements potentiels dans les besoins de leurs utilisateurs ou dans leurs environnements. Néanmoins, la synergie de ces deux paradigmes doit être canalisée et vérifiée par l'application des méthodes formelles, qui, utilisées pragmatiquement, se révèlent d'un grand intérêt dans le domaine de la vérification des systèmes complexes.
Dans cette thèse, nous proposons une approche formelle permettant au concepteur d'effectuer des spécifications et des vérifications en amont pour les systèmes logiciels à base de composants logiciels et d'aspects.

Mots-clefs

Développement basé composant, Développement orienté aspect, Méthodes formelles, DSL, Systèmes de transition étiquetés, UPPAAL.

Formal Approach for Combining aspects and Software Components

Abstract

In the present work, we assume that a synergy between component-based development and aspect-oriented development is a promising issue. In fact, each of the two paradigms can bring advantages for the other and fill its shortcomings. On the one hand, the aspects

will have the same structuring properties as the software components, thus allowing their reuse. On the other hand, the expressive power of aspect-oriented programming presents undeniable advantages, in particular for the evolution of software components, thereby allowing their adaptation to potential changes in the needs of their users or in their environments.
Nevertheless, the synergy of these two paradigms must be channeled and verified by the application of formal methods, which, used pragmatically, are of great interest in the field of verification of complex systems.
In this thesis, we propose a formal approach allowing the designer to perform upstream specifications and verifications for software systems based on software components and aspects.

Keywords

Component-based development, Aspect-oriented development, Formal methods, DSL, Labeled transition systems, UPPAAL.

Contents

Chapter 1

Introduction

1.1 Context and motivations

In recent years, software systems have become more and more complex, and changes in their needs or environments have become more and more frequent. In response to this situation, new paradigms in software engineering have emerged, among which we find: component-based development and aspect-oriented development.

Component-based development, by its main virtue - reuse - allows rapid and less expensive development, since it is based on an assembly of pre-built and certified software components.

For its part, aspect-oriented development allows an advantageous separation between the business codes of a system and the transverse functionalities, the latter being encapsulated in modules called *aspects*. Further, the aspect-oriented programming, through the weaving mechanism, offers considerable expressive power allowing very significant changes to be introduced into a system during its execution.

The search for a synergy of the two paradigms seems a promising issue. On the one hand, the aspects will have the same structuring properties as the software components, thus allowing their reuse. On the other hand, the weaving mechanism could present an efficient solution to reconfigure component based systems, thus allowing them to dynamically evolve in response to changes in their needs or in their environments.

However, this synergy can run into difficult problems, such as evaluating the scope of a reconfiguration or the preservation of some properties. Faced with this situation, formal methods present a rigorous way to verify the correctness of complex systems, firstly, a real system is specified by an abstract formal model describing only the characteristics to be verified, secondly, adjacent tools are often used to verify properties on the formal model.

1.2 Problematic

The objective of the thesis is to provide a formal support allowing to model and verify the synergy of component-based development and aspect-oriented development. In fact, we recall that each of these two development paradigms complements the other and remedies its shortcomings. Indeed, on the one hand, the aspects will be endowed with the same

structuring properties as the software components, this offers them a better reuse, on the other hand, the aspect-oriented mechanisms represent a very effective means allowing to evolve the component-based systems, however, the use of aspects can have negative effects on the system, including the violation of some properties, therefore, the main question of our problematic is how to verify two types of properties on systems combining components and aspects :

- *Behavioral properties :* Such as the absence of deadlock, both between the components of the system, as between the system and its environment.

- *Temporal properties :* One can cite as examples: the execution times of the services of the components, the mutual exclusion between the services, or the order of execution of the services.

1.3 Content of the thesis

The thesis consists mainly of two parts. The chapters of the first part (from 2 to 4) are devoted to the state of the art, while those of the second part (from 5 to 11), are dedicated to the presentation of our contribution.

Chapter 1 Introduction Presents the research context, gives a brief motivation for the thesis and introduces its main objectives. Further, a description of each chapter of the thesis is also provided.

First part The state of the art

Chapter 2 Component based development Introduces the reader to component based development. In fact, this chapter mainly presents the basic notions and concepts of this paradigm, as well as a classification of formal verification issues in this field.

Chapter 3 Aspect oriented development Introduces the reader to aspect-oriented development. First, the basic concepts of the paradigm are presented. Secondly, we outline the formal verification issues in the field, in particular those dealing with the problems of interaction between the aspects.

Chapter 4 Comparative study of similar works Presents similar works, classified into three families, namely, formal approaches for component-based development, formal approaches for the combination of component-based development and aspect-oriented development, as well as formal approaches to web services. Further, other works close to our contribution are also presented.

Second part Our approach

Chapter 5 Presentation of the approach Presents an overview of the full contribution, allowing the reader to have an overview of the full approach. In fact, this chapter first introduces the approach, then details its main steps.

Chapter 6 Case study Outline a motivating example that we used to illustrate our methodology throughout the thesis; the example shows an airline ticket booking system. In fact, this chapter gives the different subsystems with their actions, as well as the timing constraints to which the booking system is subjected.

Chapter 7 Modeling Presents the modeling of the different elements of a system based on components and aspects. Indeed, we give the model of a component-oriented architecture as well as the model of its aspect-oriented evolution.

Chapter 8 Model transformation We outline the algorithms allowing the transformation of the models presented in chapter 7 towards a network of timed automata.

Chapter 9 Verification and simulation The network of timed automata obtained after model transformations represents an input to the UPPAAL model checker, thus allowing the verification and simulation of the behavior of systems based on components and aspects. All these details will be presented in this chapter.

Chapter 10 Support tools and evaluation This chapter presents our tool, TiVA Framework, supporting the methodology of our approach, this framework is mainly composed of two tools: the TiVA DSL, allowing to specify a system based on components and aspects, and the TiVA Core, allowing to transform the said specification to network of timed automata, this paves the way to perform verification and simulation using UPPAAL tool. Further, we also present an evaluation of our approach.

Chapter 11 Comparison with the state of the art This chapter presents a comparison of our work with state-of-the-art approaches.

Chapter 12 Conclusion and perspectives This last chapter summarizes all the work of the thesis, and discusses the possible issues and perspectives for future work.

Part I

The state of the art

Chapter 2

Component-Based Development

This chapter constitutes an introduction to the component-based development paradigm. Firstly, we present the main notions as well as the basic concepts. Secondly, we will discuss the usefulness of the application of formal methods for the modeling and the analysis of component-based systems, this being part of a classification summarizing the main verification issues.

2.1 Component-Based Development

In component-based development [34, 43], the construction of a software system is reduced to an assembly of separately developed software components. This offers as advantages to reduce development costs as well as time to market. Moreover, the quality of the software systems is better, since the latter are built from tested and certified components. In addition, the maintenance and evolution stages of the system are simply a replacement of software components; furthermore, in response to changes in users requirements or in the environment, component-based systems can also be reconfigured by modifying the links of their architecture.

Software component In the literature, there are many definitions of the notion of software component, according to [34], *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition"*. Indeed, a software component interacts with its environment only through its interfaces, since it is designed without any knowledge of its environment, this offers an independence allowing its use in different contexts.

Interfaces and Assembly A software component can have two types of interfaces: on the one hand, the provided interfaces, they represent the services that the component offers, on the other hand, the required interfaces, which are the services that the component needs to accomplish its functions. The assembly of a component-based system is done by linking the provided interfaces with the required interfaces of a selection of software components; however, in order to guarantee a correct assembly of these components, the compatibility of their interfaces should be verified beforehand.
The semantics of an interface is usually specified by its signature. However, the description of an interface only by its signature is insufficient for modeling and verifying the notion of compatibility, indeed, the specification of an interface must also include the definition of the behavior, such as the sequence of service calls between components of the system,

or the timing constraints, such as the execution time of a service. As we will see in the next sections, the application of formal methods is inescapable for the verification of these issues.

Component models and component frameworks Others aspects, relating in particular to the definition of the components and their composition are specified by the component model to which the component is assigned. Indeed, the component models define a specific representation, composition modes, interaction styles and others standards dedicated to software components [45]. In addition, component models form the basis for creating *component frameworks*.
Component frameworks establish the physical environmental conditions for the execution and cooperation of components in the system, and they help also to regulate the interactions between components in execution [34].
Component frameworks can only concern physical components, unlike component models, these can be defined for the different levels of abstraction for a component [79], indeed, some component models define a software component as an execution entity, this is the case for Fractal [16] for example, while others component models define a software component as a design entity, as is the case for SOFA [76].

Instance of a software component Some component models distinguish component types from their instances, allowing the creation and the destruction of component instances at runtime, as is the case for EJB [28] or CCM [23]. Others component models like Wright [5] do not take instantiation into account.

Synchronous communication vs asynchronous communication Usually, the communication between the software components is done in a synchronous manner, as is the case for Darwin [61] and SOFA. However, in some models such as EJB or CCM, communication can be done by asynchronously sending and receiving messages.

Flat models vs hierarchical models A set of basic software components can be assembled to give a composite component. In flat component models, this composite component represents the final component-based system, as is the case for EJB or CCM. However, in hierarchical component models, such as SOFA or Fractal, the composite component may in turn be subject to composition with others components, allowing the construction of a component-based system with several hierarchical levels of components. Furthermore, in hierarchical models, we must specify the interfaces to be delegated outside a composite component to be linked to compatible interfaces in the higher hierarchical levels of composition.

Single binding vs multiple binding Some component models suppose one-to-one linking of interfaces, *i.e.,* single bindings, as in SOFA, others component models allow an interface to be linked to several others interfaces, *i.e.,* multiple bindings, as is the case of EJB and Fractal.

2.2 Life cycle of a component-based system

Component-based software systems are developed by selecting and assembling *off-the-shelf components*, instead of being programmed, this makes the lifecycle of a component-based

software system different from traditional software system; it mainly comprises the following steps:

1. *Requirements specification* : It concerns collecting, analyzing and specifying the needs of the future users of the system.

2. *Architecture specification*: The architecture of the software specifies the system in terms of abstract components of design and interactions between these components.

3. *Selection and customization of components* : First, the concrete components taken on the shelf are selected according to the software architecture, in a second step; each component must be personalized before being integrated into the new system.

4. *Integration of the system* : Integration is achieved by establishing mechanisms for communication and coordination of the various components of the final software system.

5. *Test of the system* : Various methods and tools are used to test the component-based system; in fact, it is a question of checking the properties concerning functional aspects as well as those related to the quality of the software.

6. *Deployment* : This is the installation of the software components of the system on one or more computers.

7. *Maintenance and evolution of the system* : After deployment, parts of the component-based system can be modified, due to changes in users requirements or in the environment.

The concept of software construction by *reuse* is not new, indeed, the idea was already present in object-oriented programming, it was implemented by the inheritance mechanism; the relatively recent emergence of new technologies has significantly increased the possibilities of building systems and applications from reusable components.

Furthermore, building systems based on components or building components for systems in different application areas requires methodologies and processes, including not only development and maintenance aspects, but also those relating to organizational, marketing, legal and other aspects.

2.3 Development for reuse and development through reuse

The component-based software engineering process includes two separate but linked processes via a component market. In the following we present each of the two processes:

- *Development for Reuse* : This process consists of an analysis of the application domains in order to develop commercial-off-the-shelf (COTS) components related to these domains. To complete a successful reuse of the software, standards for similar systems must be identified and represented in a form that can be easily exploited to build other systems in the domain. Once created, reusable components will be available in organizations or at the market level as commercial components.

- *Development through reuse* : this is related to the assembly of software systems from the components taken on the shelf.

2.4 The objectives of component-based development

The main objectives of component-based development can be summarized as follows:

- *Reuse* : This is the main objective of component-based development. While some software components of a large system are necessarily special purpose components, it is imperative to design and assemble components in order to reuse them in the development of others systems.

- *Independent development of software components*: Large software systems should be able to be assembled from components developed by different people, for this purpose, it is essential to decouple the developers from the components of their users, this is done mainly through the specifications of the behavior of components.

- *Software quality* : A software component or a component-based system should have the desired behavior. Quality assurance technologies for component-based software systems are currently relatively premature, as the characteristics of component-based systems differ from those of conventional systems.

- *Maintainability* : A component-based system should be built in a way that is understandable and easy to evolve.

2.5 The contributions of component-based development

The contributions of component-based development can be presented as follows:

- *More efficient management of complexity*: The division of large and complex systems into sub-systems offers greater control over their complexity.

- *Time to market is reduced* : Component-based development consists of assembling existing components, which reduces development time, and therefore accelerates the time to market.

- *Costs are reduced* : While some software components are completely specific to a given application, other software components can be reused and shared with other developers, thereby reducing their costs by damping through a large population.

- *Quality is improved* : Component-based development greatly improves the quality of the systems, since the latter are built from components that are already tested and certified.

- *Easier maintenance and evolution* : The maintenance and evolution of component-based systems is easier, since most of the time they are reduced to simple additions, deletions or replacement of software components.

2.6 Classification Of Formal Verification Issues For Component-Based Systems

Formal approaches are rigorous methods aimed at modeling and analyzing complex systems. The idea of verifying programs is not new; in fact, it dates back to the 1960s. Today, formal techniques and tools are widely used in both the academic and the industrial worlds.
In our context, formal methods are essential for component-based development because they enable addressing important verification issues throughout the lifecycle of a component-based system. In the remainder of this section, we will detail these verification issues which we have classified into three levels, namely, at an individual component, during the composition of the components, and finally at the evolution level.

2.6.1 Component level

This level of analysis addresses the verification of an individual component before its composition with the rest of the system; we classified this verification into two types:

- *Context-independent verification* : it consists of verifying the properties of a component in the isolation, thereby independently of its deployment context; indeed, the issues to be checked can concern the absence of deadlock in its own specification or the coherence of the specification of its temporal constraints.

- *Context-dependent verification* : In component-based development, components are developed independently of their deployment context; therefore, component correctness can be very difficult to define, as a component may behave correctly in a context but incorrectly in another. Existing approaches remedy this situation in two different ways; some approaches [65, 66] propose to attribute to each component a description of its properties, thereby enabling the user of component to decide if the latter can behave correctly in a given context. Other approaches [90, 22] deliver software components with a set of quality properties that are guaranteed in all contexts satisfying a number of conditions.

2.6.2 Composition level

Compatibility of components

The software components constituting a component-based system can be delivered by different sellers; therefore, verification of their compatibility is an important issue. Some approaches define compatibility only in terms of signatures of services linking components [89, 23, 60]. However, this description is by no means exhaustive, because it does not include for example, the specification of the services calls sequence of a component, such an aspect is more a matter of behavior. On the other hand, other approaches offer a richer description of compatibility, including description of the behavior [26]. This makes it possible to verify that the composition will not lead to an erroneous interaction between the components of the system.
Some approaches propose to verify compatibility at design time, while others perform checks during execution, thereby detect bad interactions between components dynamically, using a test environment in which the concerned components are duplicated [71].
Moreover, even if the components are not completely incompatible, they can sometimes cooperate correctly by generating appropriate adapters of their interfaces. Some approaches generate adapters for connecting components belonging to different component models [35, 25]; this can be done in a fully automatic manner. Other approaches include adapters for integrating an incompatible functionality of components [38], in which case, additional input is required from the user or the monitoring phase to provide information concerning the parts corresponding to the incompatible functionality.

Assembly of components

The process of assembling components is mainly twofold: identifying the correct components taken on the shelf, and their connections together, so that the resulting component-based system corresponds to the desired requirements.
Usually, assembly strategies focus on finding the most cost-effective solution with respect to time [38]. The cost function can, for example, evaluate the components in terms of their performance measurements or the minimization of new requirements generated by the added components. The assembly can be selected based on an exhaustive evaluation of all possible alternatives [9], or via an iterative construction of a relatively optimal solution [29].
In this context, formal methods make the problem of assembly of components considerably simpler by simply providing a design of the component-based system comprising specifications of a set of components and their connections, the problem being reduced to simply finding the correct component implementations taken on the shelf and formally verifying their compliance with the expected specifications.

The global verification

Formal methods are very useful for verifying the global properties of a final component-based system. In this case, formal analysis generally includes:

- Verification of standard coordination errors.
- The absence of deadlock in the system.

- Verification of the different timing constraints in the global system.

- The order of execution of a set of services of a components selection in the final system.

- Verification of the number of components that can simultaneously access to the same service.

This verification can be carried out on the whole of the final component-based system or simply on a well-defined part.
Furthermore, in addition to checking properties, formal methods can also help in optimizing component-based systems, namely:

- Detection of inactive components, which can be removed from the system.

- The search for optimal system deployment by placing components in compute nodes based on the density of interaction between them [95].

As with compatibility, some approaches check the properties of a global system at design time, while other approaches allow dynamic verification of the system, in fact, the conformance of the current behavior of the components in execution is verified in parallel with its specification [72], thereby any errors are reported in case of discrepancy.

2.6.3 Evolution level

After the deployment phase, a component-based system can evolve or adapt, in response to changes in users needs or changes in its environment [88], namely: interoperability with others systems, optimization of computational algorithms, or technical changes.
Formal methods and techniques are very useful for modeling and analyzing the evolution of component-based systems [96]. We have classified this analysis into two types:

- *The dynamic reconfiguration of the architecture* : this mainly includes the change of the links between the system components as well as the creation and destruction of the instances of the components. At this level, formal analysis seeks to verify the coherence of the global system after a dynamic reconfiguration.

- *Substitutability* : one or more components can be replaced with new ones. Generally, approaches addressing this issue define an equivalence relation between the old and the new component, in order to verify that the substitution does not violate the correctness of the global system [73]. However, in some cases, the verification of the equivalence between the two versions of the system is not necessarily strong, because it is only necessary that the new system satisfies a given explicit property, this is considered much more by the approaches that do not aim to guarantee that the behavior remains unchanged, but rather to identify the behavioral differences between several versions of the system [62].

Furthermore, the evolution of a component-based system is usually defined with a set of evolution rules.

2.7 Conclusion

In this chapter, we wanted to present the principles as well as the basic concepts of component-based development, before proposing a classification of the main formal verification issues for software components.
Thus, the formal methods and techniques, used pragmatically, are of great interest for component-based development, both for its analysis in isolation, and when it is combined with other paradigms, in our case, the aspect-oriented development, which we will present in the next chapter.

Chapter 3

Aspect-Oriented Development

Aspect-oriented programming is positioned at the limits of object-oriented programming, in fact, it offers better management of the complexity of software systems and facilitates their evolution. In this chapter, we will introduce the basic principles and concepts of aspect-oriented programming, then we will explain the usefulness of aspect-oriented design in the development cycle of complex systems. Finally, we will describe some problems related to this programming technique, in particular, those having to do with aspect interferences.

3.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [53] is a programming technique proposed by Gregor Kiczales in 1997, it is one of the implementations of *advanced separation of concerns* [15]. In this approach, a software system is considered to be composed of a functional part and a non-functional part. The functional part relates to the business part of the system, in other words, it is the functionalities for which the system was designed. The non-functional part represents concerns that are independent of the main mission of the system but are necessary for its functioning, as an example of non-functional concerns we can cite: security, synchronization, persistence management, transactions management, and optimization. Moreover, these concerns are *crosscutting*, since each non-functional concern crosscut several functional classes.
In the aspect-oriented approach, each of these concerns is called an *aspect*, the objective of the approach is to factorize these aspects outside the functional part, compose and reuse them in an efficient way [53].
In general, aspect-oriented programming has the advantage of reducing the complexity of a system, improves its understanding, reduces its code, and improves its evolution and maintenance. In addition, modeling is closer to our perception of the real world.
In practical terms, AspectJ [51] represents the first concrete realization applied to the Java language, it remains today widely quoted in all the works dealing with the aspect-oriented approach and remains considered the reference, in particular, for its language of pointcuts.

3.2 The need for aspect-oriented programming

Aspect-oriented programming is positioned to the limitations of object-oriented programming. In fact, in the practice of object-oriented programming, it can be seen that each functionality is not perfectly embodied by a class or by a restricted group of classes, in-

deed, it is common to see in the functional code of a system several non-functional concerns independent of this code, this situation poses mainly two problems, which are, the entanglement and the scattering of the code [53]:

- *Tangled code* : Within a given class, we find both the code implementing the functional part of the system as well as the code implementing the non-functional part (*i.e.,* the concerns); this mixture prevents the reuse of the classes.

- *Scattered code* : For a given concern, such as security, for example, one finds that its code is duplicated in several classes in the system; therefore, changing the security policy requires a series of changes in several classes. The scattering of the code makes the maintenance and evolution of the system difficult.

The aspect-oriented programming modularize the code of each non-functional concern in one aspect, outside the classes; these will contain only the code relating to the functional part, so the two problems of entanglement and scattering of the code are eliminated.

3.3 Basic principles and concepts

The aspect-oriented programming make very powerful transformations in a basic program (for example: an object-oriented program) by a behavior injection mechanism called *weaving*. The behavior is encapsulated in a module called *aspect*. The aspect contains mainly two types of elements: an *advice*, representing the code to be injected, and a *pointcut*, specifying the locations of the injection of this code into the base program, these locations are called *join points*. A join point can be for example a method call. Depending on its type, an advice of an aspect can be executed *before*, *after* or *around* a join point in a basic program.
In the following, we present in more detail the basic principles and concepts underlying aspect-oriented programming:

- *Dependency inversion principle.* This notion makes it possible to change the direction of the dependence between two entities, where at the beginning, the least abstract (*i.e.,* lowest level) depends on the more abstract (*i.e.,* higher level). For example, for the persistence management concern, without the inversion of dependencies, the program must retrieve the reference to the programming interface and call it, in other words, the implementation part of a program depends on a higher level interface (*i.e.,* persistence management). The aspect-oriented approach is often considered one of the implementations of the dependency inversion principle.

- *Advice.* The advice implements the behavior of an aspect; moreover, the latter can be composed of several advices. An advice is a construct that looks like a method in object-oriented programming; it is used to declare a code that must be executed once the join points expressed by a pointcut are reached. There are three types of advices: the *before* advice, the *after* advice and the *around* advice. For example, if a pointcut is a method call, the concerned aspects must contain a pointcut that specifies the join point: method call, and an advice that will be executed before,

after or around this method call. In the following, we present a brief description of the three types of advices:

- *The before advice*: It executes its behavior before the execution of the method call in the base program.

- *The after advice*: It executes its behavior after the execution of the method call in the base program.

- *The around advice*: Its behavior is executed before and after the method call, in that case, the first part of the behavior of the advice is executed, afterwards, the processing associated with the method call in the base program is relaunched by a special instruction called: *proceed*. Finally, the advice executes the second part of its behavior. Furthermore, the around advice can override the execution of the method call, in that case, the use of the proceed instruction is omitted.

The power of the advices lies mainly in the use of wildcards (*) in the expressions of the pointcuts, instead of a type, a method, a parameter or part of a name. Moreover, the descriptive power of an advice also comes from the fact that they can access the dynamic contextual information captured by the join points, this is known as the *context exposure*; for example, pointcuts can capture parameters and the target object of a call.

- *The base.* The base corresponds to the program without aspects, for example, in object-oriented programming, the base represents the functional classes of the system, on which non-functional concerns (*i.e.,* aspects) will be woven.

- *The aspect.* An aspect is a module encapsulating the behavior of a crosscutting concern, it is a construction that looks like a class in object-oriented programming; indeed, it contains fields and methods. An aspect mainly contains pointcuts and advices. Moreover, an aspect may also contain *introductions* or *inter-types declarations*, which allow the declaration of new members that can be inserted into the specified classes, for example, the addition of a public or private method, the addition of a field, or the declaration of an interface implementation in a class [91]. Furthermore, it is also possible to modify what a class extends or implements.

- *The weaving.* This operation looks like a compilation, the principle of weaving of an aspect to a set of classes consists in assembling these entities in such a way that the result is the basic system (*i.e.,* functional classes) extended by the behavior of the aspect, in other words, it is a question of integrating the classes and the aspects in order to produce an executable final code of the system. Moreover, the aspects weavers may be static or dynamic; in the first case, the operation is performed during the compilation phase; in the second case, it is performed during execution. Furthermore, AspectJ [51] also allows weaving aspects when loading the classes by the Java Virtual Machine.

- *A join point.* A join point is a well-defined point in the execution flow of a base program; it represents a point of interruption where an aspect can be executed. A join point can be for example: an invocation of a method, its execution and the return of values, an exception block, or the access or modification of the field of an object. Indeed, the execution of the base code is interrupted at a given join point, to allow the aspect code to execute and realize the purpose of the concern it implements, afterwards, the base program resumes its normal execution.

- *A pointcut.*A pointcut is used in the source code of an aspect to describe a set of join points; it allows specifying where an advice applies. A pointcut is an expression that uses boolean operators and specific primitives to capture a set of join points, as well as information about their dynamic context. For example, a pointcut can specify as a join point: a method execution, as well as the parameters of the call of this method.

3.4 The aspect oriented approach in design

The simple detention of the best methods and tools of aspect-oriented programming such as AspectJ, does not automatically give the possibility of applying them with ease, in case the software system to be developed is designed without the consideration of the aspect oriented concepts. In fact, the programmer will inevitably have to re-design the few parts of the system concerned by the introduction of the aspects. Furthermore, the change of design is often not explicitly the responsibility of the programmer.
Therefore, the aspect-oriented approach should be taken into account at the early stages of the software lifecycle. In general, like the structured software life cycle or the object-oriented software lifecycle, an aspect-oriented software life cycle is required.
In the remainder of this section, we first present the disadvantages of using aspect-oriented programming without an aspect-oriented design. In a second step, we will outline the two main approaches for considering aspects in design.

3.4.1 The disadvantages of using aspect-oriented programming without an aspect-oriented design

The aspect-oriented approach offers undeniable benefits in programming a system by separating concerns, reducing costs and time to market, increasing reuse and facilitating software evolution. However, without an aspect-oriented design, the designer delivers a specification for the programmer, and then the latter will often be forced to change the design of the system in order to introduce the separation of crosscutting concerns at the programming level. This has the following negative consequences [21]:

1. Since the programmer has changed (part of) the design, the consistency with the entire original design can no longer be guaranteed, the programmer only has a partial view of the whole system, he does not know if the changes he introduced are compatible with other parts of the system or not. In addition, the problem could be even worse if other programmers also introduce changes.

2. Due to the changes introduced by the programmer, the implementation of the system no longer corresponds to the original design; therefore, there is a lack of consistency

between the implementation and the design. As a result, the maintenance and evolution of the system become difficult, moreover, we do not know how are reflected in the design, the changes related to an aspect introduced by the programmer at the implementation level. Thus, the benefit of an easier software evolution, one of the main demands of the AOP is reduced.

3. Using AOP without an aspect-oriented design, the task of the programmer will be more difficult than the use of conventional techniques. In fact, in addition to his normal task, he is forced to redesign the system implicitly. Thus, the enhanced programmer's productivity benefit, one of the claimed characteristics of the AOP is reduced.

4. Using AOP without an aspect-oriented design, the programmer assumes the responsibilities that clearly belong to the designer; this can lead to conflicts in the case that errors occur in the system. Indeed, it will not be known who is responsible for such a problem, the programmer or the designer.

In conclusion, aspect-oriented programming complements current programming techniques such as object-oriented programming by offering a new concept: the *aspect*, the latter captures the crosscutting concerns in a modular way. However, in order to use AOP in real software projects, and involving multiple programmers, aspect-oriented design techniques must be provided.

3.4.2 Aspect oriented methods and tools in design

The aspect-oriented design methods and tools allow the software architect to provide the programmer with an accurate procedure describing the management of the separation of concerns in the system. This makes it possible to have a correspondence between the implementation and the design, thus, the problems declared above may be resolved.
To achieve this, two main approaches are proposed, which allow the introduction of the aspect-oriented approach in the design [21]:

- An *« extension to UML »* : This approach is relatively mature, it is an informal technique that extends UML [13] with an aspect profile, in this profile, each aspect is represented with a new stereotype; moreover, a notation to describe the behavior of the aspect-based system is also provided. However, with UML, the means of making formal specifications are relatively limited; in fact, this is a real need for different parts of several systems.

- An *« extension to architectural description languages »* : This approach can be considered as an ongoing field of research, which nevertheless produced significant results; in fact, it concerns the issues of integration of aspects at the architectural level. Indeed, architectural description languages (ADLs) [4] allow software architects to specify the functionalities of a system by means of components and interactions between these components through connectors. Basically, ADLs do not provide primitives for specifying aspects. In order to specify separately such concerns, new

elements are required in the ADL.

The capabilities supported by AO-ADL (aspect-oriented architectural description languages) can be summarized as follows:

1. *Specification of components with interfaces and connections between interfaces.* This functionality is already provided by classical ADLs; however, the AO-ADL introduces in addition, new primitives allowing specifying the join points. The join points specify locations in the architecture in which the behavior of the crosscutting concerns must be woven.

2. *The specification of aspects.* The aspects are the new elements introduced by the AO-ADL, they allow modularizing the behavior of the crosscutting concerns separately. While an aspect encapsulates the behavior of a functionality of the system, it has a different structure than a component; indeed, it is described with new primitives. In addition, an aspect specifies a role that matches the different join points it affects. The ultimate goal is to specify "what", "when" and "how" the behavior of an aspect should take place at a given location in the architecture.

3. *Specifying connectors between join points and aspects.* The purpose of these connectors is to link the overall specification representing the whole system; indeed, the connectors specify "how" and "where" each aspect should be woven. In fact, this scheme is not new, models and coordination languages already solve similar problems [27], in particular, exogenous coordination models such as those presented in [68] and [7], the latter define two types of components: functional components, specifying system functionalities, and coordination components, specifying how the functional components should coordinate. In addition, the coordination components determine "where" and "how" the actions implemented by the functional components are to be executed. In our case, the functional components represent all the components and aspects; on the other hand, the coordination components represent connectors between the components and the aspects, these connectors determine two elements: first, how the behavior of the aspect is woven in the join points of the components, in other words, via a synchronous or asynchronous mode, secondly, under what conditions the aspects must be woven.

3.5 The contributions of aspect oriented development

The benefits of aspect-oriented programming are summarized in [55] as follows:

- *Responsibilities are clear for each module.* In AOP, each module encapsulates the behavior of a well-defined crosscutting concern; therefore, the module is solely responsible for this concern and not other concerns in the system. For example, a module that accesses a database is not also responsible for establishing connections. This clear assignment of responsibilities for each module improves traceability.

- *High modularization.* The AOP offers a mechanism for addressing each concern separately; indeed, the source code is modularized even in the presence of crosscutting concerns, thus the AOP ensures minimal coupling. This offers the advantages of lowering the duplicated code and providing an easier implementation to understand and maintain.

- *An easy evolution of the system.* The AOP modularizes in an independent manner the aspects of the system on the one hand, and the code modules on the other hand. Consequently, the addition of a new concern is reduced to a simple integration of a new aspect without the core modules being changed. Moreover, adding a new module to the core does not take into account the existence of the aspects. The overall effect is a coherent evolution and a faster response to new needs.

- *Delayed design decisions.* With the AOP, the architect can delay decisions about future requirements, as they can later be implemented as separate aspects and without large changes in the system. As a result, the architect focuses only on the needs of the current core of the system. In fact, the implementation of a characteristic relative to a probable future need can generate additional efforts and costs if this future need is not met. Delaying design decisions accelerates time to market, reduces costs, and allows more efficient processing of the needs of the current core.

- *Improved code reuse.* The AOP implements each aspect as an independent and separate module; therefore, the modules are weaker coupled than in the case of equivalent conventional implementations. Indeed, the core modules are not aware of the modules encapsulating the aspects, only the modules relating to the specification of the weaving rules are aware of all the couplings in the system. As a result, the change in the configuration of the system is reduced to a simple change in the specification of the weaving, without having to change the modules. For example, a database module can be used with a different security policy, by making simple changes in the weaving rules, and without having to change this module. Low coupling provides better reuse.

- *Faster time-to-market.* Delaying design decisions involves a faster development cycle. More reuse of the code leads to a reduced development time. An easier evolution allows a faster response to new needs. All this leads to faster systems to be developed and deployed.

- *Reduced implementation costs.* Development by reusing existing modules reduces the cost of implementation. Furthermore, the integration of crosscutting concerns does not require modification of several core modules, as is the case in conventional processes; this makes the implementation even cheaper.

3.6 Problems of aspect interferences

Aspect-oriented programming offers mechanisms for making very powerful changes in a system. However, this expressive power should be rigorously controlled, as aspects can

have a negative effect on the system. In fact, the problems that can arise in this context mainly concern the interferences of aspects [86]; the latter can be classified into two types:

1. *Aspects/base interferences.* This problem can be caused by the use of wildcards in the expressions of pointcuts, in case of errors, behaviors of aspects can be woven into inappropriate locations, this can cause negative effects in the system. For example, aspects can modify variable values, and can therefore change the flow of control of the execution of the base system.

2. *Aspects/aspects interferences.* This type of problem can occur if two conditions are met:

   (a) Several aspects must be woven in the same join point.

   (b) One or more possible constraints between these aspects are not respected at the moment of their weaving.

   In this case, conflicts may arise between these aspects. Among the constraints that may exist between the aspects we can cite:

   - *Execution order of aspects* : Sometimes, not executing a number of aspects in a specific order can lead to the violation of certain properties of the basic system or of other aspects. Moreover, this order of execution of the aspects can be dynamic; in fact, it can change according to the dynamic state of the system or the context in which the aspects will be woven.

   - *The dependency between aspects* : This is the case where the execution of one aspect requires the execution of another.

   - *Mutual exclusion* : This constraint expresses the situation where several aspects do not have to be executed simultaneously on the same joint point, since for example, they have the same effect, or they have contradictory effects.

The interferences of the aspects are also called the *interactions* of the aspects. Currently, formal methods and tools represent a promising issue to address these problems.

3.7 Conclusion

The aspect-oriented approach first emerged in the programming phase, it allows a more efficient implementation and facilitates the maintenance of the code, however, the developers are still relatively reluctant to its use, this is mainly due to a lack of methods and tools to take into account the aspect-oriented approach in the early stages of the system life cycle. To remedy this, aspect-oriented architecture description languages combined with the use of formal methods present a very promising solution to meet this need.

Chapter 4

Comparative study of similar works

In this chapter, we will present a comparative study of approaches in the field of formalization of software components and aspects. Indeed, we have classified these works into three main families:

- Formal approaches for component based development.
- Formal approaches for the combination of component-based development and aspect-oriented development.
- Formal approaches to component-based development applied to the domain of web services, in fact, web services being a typical application domain of software components.

Further, we will also present other works having points in common with our field of research.

4.1 Formal approaches for component-based development

This section is devoted to the study of the modeling and analysis approaches of component-based systems. First, we will present the most referenced works in this field, secondly, we will give a comparative study of the approaches.

4.1.1 Wright : A formal basis for architectural connection

Wright's authors [4] start from the observation that software systems are becoming more and more complex, which makes software architecture a crucial design problem. The objective of this work is to arrive at a formal basis allowing to describe and analyze the design of software, in fact, the approach deals with a particular aspect of the design: the interactions between the software components.
The main idea of the approach is to define the connectors of the architecture as explicit semantic entities, they are independent of the interfaces of the components and have their own semantic definition, consequently, the connectors are considered as types and can be

instantiated.
The specification of a connector is divided into two types of elements: roles and glues. A role defines the behavior of a participant (*i.e.,* component). Glues coordinate and manage constraints between roles, in other words, glues specify the way in which components interact in the architecture.
In Wright, compatibility checking in software architectures is performed in the same way as type checking in programming languages.

4.1.2 Formal modeling and analysis of component-based systems

Zimmerova in her thesis [94] proposes to model and verify the interactions between components in component-based systems, for this, the work is based on the approach of correctness by construction.
For each stage of the lifecycle, a model corresponding to the current version of the system is produced. To build a model, a formalism based on automata is created for each service of a software component, thereafter, the automata relating to the services are composed together in order to produce a single automaton representing a software component, these automata are composed in turn, thus, giving an automaton representing a composite component; finally, the automata of the various composite components are composed together allowing to have a single automaton representing the full system. Further, the composition of the automata in the different stages is done using one of the four composition operators: the handshake composition, the star composition, the full composition and the assembly guided composition. The choice of an operator depends on the context of the composition.
On the model representing the full component-based system, formal verifications can be performed using the equivalence of the observation, this is a technique based on the bissimulation [59]. Verification consists of making comparisons between the different models of the system relating to the different stages of its life cycle.
This work allows, among other things, the verification of the substitutability of one component with another, this results in the verification of the equivalence of the model of the implementation of the new component with that of the old one.

4.1.3 A formal approach to component adaptation

The objective of this work [14] is to provide a formal method allowing to adapt heterogeneous software components in an architecture.
For this, the approach includes the behavior in the interfaces of the components, indeed, the authors propose a high level specification allowing to express the specification of adapters between the software components, thereafter, a complete automatic procedure is provided, which allows the concrete derivation of adapters from their high level specifications.

4.1.4 A formal approach of dynamic reconfiguration

This work [83] offers a formal support for modeling and verifying dynamic introspection and reconfiguration in component-based systems.
The approach is specific to the Fractal component model [16], so at the same time it represents a formal specification that overcomes the shortcomings of the informal Fractal specification.
The basic Fractal primitives are specified in a language called FracL, further, a scripting language has been proposed for writing reconfiguration programs using Fracl. In addition,

Table 4.1: Formal approaches for software components. Part 1.

Approaches	General or Specific to a component model	Flat or Hierarchical
TOSEM'97 [4]	General	Flat
Zimmerova (Thèse) [94]	General	Hierarchical
JSS'05 [14]	General	
L'OBJET'08 [83]	Fractal	
ICSC'06 [8]	Kmelia	Hierarchical

Table 4.2: Formal approaches for software components. Part 2.

Approches	Formalism	The verified issue	Tools
TOSEM'97 [4]	CSP	Compatibility	FDR
Zimmerova (Thèse) [94]	LTS	La substitutability	DiVinE
JSS'05 [14]	Mu-calculus	Adaptation of components	
L'OBJET'08 [83]	LTS	Reconfiguration	Focal
ICSC'06 [8]	LOTOS	Composability	LOTOS CADP

the model has been translated into a proof tool called Focal.
This work is specific to a particular component model (*i.e.,* Fractal), however, it is technology independent as Fractal is independent of any programming language.

4.1.5 Checking component composability

In this work [8], the verification of the composability of components is reduced to the verification of the composability between their services. For this, a component model called Kmelia has been proposed, an associated formalism is presented, in fact, the authors seek to enrich the interfaces of the components and model the behavior of the services of the components by extended labeled transition systems. Thus, the verification of the correctness of the components is reduced to the verification of the behavioral compatibility and the composability between the links of the services of the components, in other words, the approach verifies the static interoperability and the dynamic interoperability at the level of the interaction of components.
On a practical level, the components as well as their services are translated on the LOTOS formalism, and experiments were carried out in the LOTOS CADP toolbox.

4.1.6 Comparison of approaches and discussion

The tables 4.1 and 4.2 compare formal approaches to component-based development according to five criteria.

In the table 4.1:

- Criterion 1: shows whether the approach is general or specific to a particular component model.

- Criterion 2: specifies whether the approach is limited to flat systems or also supports

hierarchical component-based systems.

In the table 4.2:

- Criterion 1: specifies the formalism used for the modeling of a system.
- Criterion 2: gives the verified issues by the approach.
- Criterion 3: gives the different tools used in the work.

Most formal approaches for component based systems model components as first class entities, as is the case with the approach [83] for example, but in reality this depends on the issue that the authors seek to verify. Indeed, in wright [4] for example, it is the connectors which are considered as first class entities since the main objective is the study of the compatibility between the components. On the other hand, the authors of Kamelia [8] offer a detailed study of the composability of services, therefore, it is the services that are considered as first-class entities, and not the components.

Although the [4], [14] and [8] approaches address verification differently, in general, their authors seek to verify the interactions between software components that have been developed separately, in order to adapt their collaboration in the case of incorrect interactions.

The approach [83] seeks to verify mainly a rather architectural problem, in fact, properties of the system are verified after the change of components and/or the links between the components of the architecture, in response to a change in environment or in user needs.

The approach [94] addresses the verification of the evolution of the system by the replacement of components, further, other issues of verifications are also addressed, namely, the verification of the equivalence between the models of the architectural design and the model of the specification, or the verification of compliance, *i.e.,* the equivalence between the specification and the implementation.

4.2 Formal approaches for the combination of component-based development and aspect-oriented development

The combination of the component-oriented paradigm and the aspect-oriented paradigm represents a promising issue [44], in fact, each of the two paradigms brings advantages for the other and remedies its shortcomings. In what follows, we will present a reminder of the benefits of each paradigm for the other:

The benefits of software components for aspects. The application of the principles of component-based development on the aspect-oriented approach allows, among other things, to endow the aspects with the same structuring properties as the software components and to apply the principle of off-the-shelf components, this mainly offers a better reuse of the aspects.

The benefits of aspects for software components. Similar to the object-oriented approach, the component-based approach also suffers from code entanglement and scattering problems. Faced with this situation, the use of aspects eliminates these two problems for the components. Further, from a practical standpoint, component-based systems use the expressive power of aspects as a solution to adapt to changes in the environments and changes in the needs of their users.

4.2.1 The need for formalization

Although the search for synergy between aspects and components brings certain advantages, aspects can also have negative effects on some properties of software components. For example, following their application in inappropriate places. To cope with this situation, one of the most reliable ways to control and channel this synergy is to resort to formal methods.
In the rest of this section, we will present the formal approaches to combining components and aspects, classified into two families: inter-component approaches and intra-component approaches. Subsequently, we will give a comparative study of these approaches.

4.2.2 Classification of formal approaches for combining component-oriented development and aspect-oriented development

Although the search for a synergy between component-oriented development and aspect-oriented development is a promising issue, relatively little works is currently dedicated to this area of research. In what follows, we will present a classification of formal combination approaches into two families: *inter-component approaches* and *intra-component approaches*.

Inter-component approaches

The approaches of this family consider software components as black boxes, in fact, these approaches combine components and aspects by weaving aspects at the inter-component level, in other words, aspects do not have access to internal details of components, on the other hand, weaving locations (*i.e.,* join points) can only relate to interactions between components. For example, an aspect is woven before a service is called between two components.

PRISMA : Designing software architectures with an aspect-oriented architecture description language The PRISMA approach [74] proposes a formal model making it possible to integrate aspects into ADLs (*i.e.,* Architecture Description Languages), indeed, aspects are considered from the first stages of the development cycle (*i.e.,* the definition of needs and design), this provides consistency between the different models of the system, and consequently allows to keep a trace between the architectural model and the code. Further, the fact that all properties are introduced at the architecture level and not at the implementation level, makes the approach independent of technology.
Unlike other approaches, an aspect in PRISMA is not simulated by another term, such as component or connector, but rather it is introduced as a new concept and considered as a first class entity, this allows the reuse of aspects and facilitates their maintainability.
In addition to the classic reuse of components, PRISMA also allows the reuse of aspects, connectors, interfaces and the full system, this is due to the fact that its language is divided into two levels of abstraction: the level of definition of data types and the configuration

level. In the first level, we define the different types of system entities such as components, connectors, aspects and interfaces. In the second level, for a given system, we create a configuration by instantiating the types defined in the first level.
Further, PRISMA allows automatic generation of code from the PRISMA AOADL specification (*i.e.,* Aspect-Oriented Architecture Description Language), which saves development time.
Unlike formal verification approaches at design, PRISMA allows dynamic weaving of aspects, and allows verification of system evolution at run time.

Property-preserving evolution of components using VPA-based aspects The objective of this work [70] is to verify two main properties on software components: compatibility and substitutability after the adaptation of a component-based system by the weaving of aspects.
In this approach, the interactions between the components of a system are managed by the interaction protocols, thus, the aspects are woven on these protocols. In fact, the work presents an extension of an earlier work on the use of VPA (visibly pushdown automata) based protocols for software components, the work has been extended to verify properties after aspect-oriented adaptations of these components.
Visibly pushdown automata are more expressive than finite state machines, but at the same time, easier to use than pushdown automata. Usually, the approaches use finite state machines, nevertheless, the advantage of using a stack is the evolution in recursive contexts where we express the nesting of calls as is the case for example in P2P algorithms. Further, the formal verification can be performed by the theorem proving or by the model checking.

Understanding aspects via implicit invocation The authors of this approach [92] assume that aspect-oriented programming offers significant advantages through its expressive power, however, its application could break the fundamental principles of component-based system design. As a solution, the authors propose to map the aspect-oriented programming on the implicit invocation (*i.e.,* II), in order to establish a correspondence between the two paradigms, this allows to use the models and the tools which were already realized in the domain of the implicit invocation to check the adaptation of the software components by the aspects.
Further, by this connection between II and AOP, the authors aim to offer a better theoretical understanding and practical use of aspect-oriented techniques, in particular, the exploitation of the existing model checking of the explicit invocation for aspect-oriented techniques.

Intra-component approaches

The approaches of this family are not limited to the inter-component level, but allow the weaving of aspects even inside the components, by the interception of the internal join points to the components. The risk of such a practice is to break the encapsulation and certification of components that are already tested and closed.
In other words, this family seeks to offer more flexibility in order to benefit from a greater expressive power of aspect-oriented programming, provided that the internal correction of the software components is preserved, by having recourse to the use of formal methods.

A Formal Model of Modularity in Aspect-Oriented Programming Jonathan Aldrich in [3] proposes a combination approach allowing to weave the aspects inside the software components. To increase the reasoning on this weaving, the author extends a previous work (*i.e.,* the approach of open modules [2]) with a new language called TinyAspect, the latter represents a formal support allowing to prove the correctness of the system after its evolution by aspects.
The open modules approach consists in adding to the classic interfaces of the modules, a set of join points representing the internal calls on which the weaving of aspects can be done. Further, on the one hand, the developer is not required to know the details of the aspects that can be woven, and on the other hand, the customer is not required to be aware of the components to weave its aspects; only the developer of a module as well as the client must just respect some rules, which constitutes a contract for them.
This approach allows on the one hand, several weavings of aspects without touching the properties of the components and their certification, on the other hand, it allows to make changes in the components without affecting the customers (*i.e.,* the aspects). Further, developers and customers may not follow open module rules, this may provide better flexibility as well as better reuse, however, no certification is guaranteed on the system.

Validation of context-dependent aspect-oriented adaptations to components The objective of this work [24] is to verify the aspect-oriented adaptations of software components, to their deployment, composition and execution contexts.
Similarly to the approach [3], the authors seek to increase the reasoning on intra-component weaving, in order to preserve the certification of software components, which results in a certain number of properties, which are: semantic contracts, time constraints and quality attributes. To do this, the authors propose to associate a profile for the component to be adapted, as well as a profile for the aspect to be woven, describing their essential properties, in order to finally be able to derive the expected profile of the woven component, and compare it with its actual profile resulting when running the real system.

4.2.3 Comparison of approaches and discussion

The tables 4.3 and 4.4 give an overview of the approaches and compare them according to five criteria.

In the table 4.3:

- Criteria 1 and 2: allow us to see if the approach is limited to the weaving of inter-component aspects and/or even addresses the weaving of intra-component aspects.

- Criterion 3: indicates the availability of the tool supporting the approach. Note that, the notation (x) shows that the authors speak of a tool but do not give more details.

In the table 4.4:

- Criteria 1 and 2: specify the phase of the verification: during the design or during the execution of the system.

Table 4.3: Formal approaches for the combination of components and aspects. Part 1.

Approaches	Inter-components	Intra-components	Tool
ICASE'06 [74]	x		x
MMIC'07 [70]	x		(x)
ICASE'04 [92]	x		(x)
ICAOSD'11 [3]		x	x
WCOP'04 [24]		x	

Table 4.4: Formal approaches for the combination of components and aspects. Part 2.

Approaches	Design	Runtime	Specific language	Symmetry
ICASE'06 [74]		x	x	x
MMIC'07 [70]	x			
ICASE'04 [92]	x			
ICAOSD'11 [3]		x	x	
WCOP'04 [24]		x		

- Criterion 3: indicates whether the approach offers a new language.

- Criterion 4: shows whether the approach considers aspects as first-class entities as it does for components.

According to the table 4.4, PRISMA [74] represents the only approach which considers the symmetry between the components and the aspects in a system, this allows to apply the structuring properties of the software components on the aspects, which offers better reuse of aspects.
As indicated previously in [21], the authors propose a classification of approaches integrating aspect-oriented principles from the first stages of development, into two families: an "extension to UML" and an "extension to architecture description languages". PRISMA is part of the second family. The integration of aspect-oriented principles from the first phases of the development cycle, mainly allows to keep a consistency of the view of the system throughout its full development cycle, the developer will not have to change the design during the implementation, and the responsibilities of the designer and the programmer are clear and separate. However, similar to the approaches [70] and [92], PRISMA does not address the checking of the weaving of aspects within components.
The authors of [92] propose to reuse the methods and tools of implicit invocation for aspect-oriented programming, this makes it possible to make profitable the efforts invested in the implicit invocation, however, this situation shifts the complexity on the side of the user, this one will have to provide an additional effort in learning a new domain (*i.e.,* the techniques of implicit invocation). Further, the authors of [92] do not present the algorithms of translation from II to AOP. On the other hand, the proof that the model in II expresses reality in AOP is not provided in the work, and proof that the properties checked in II tools reflect the correct properties in AOP, is also not given.
The main advantage of the approach [3] is that it is based on the principle of abstraction, in other words, clients (*i.e.,* aspects) do not affect modules and at the same time do not depend on their internal details, this allows separate development of components and aspects. However the authors of [3] require the learning of a new language, which supposes

an additional effort on the side of the user. Further, setting join points at interfaces, on the one hand, limits the reuse of the module, and on the other hand, creates a certain dependence between the module and the wearable aspects, this prevents the evolution and poses *the paradox of AOSD evolution* [84]. However, the authors suggest a use without respecting the rules, but in this case correctness of the system is no longer guaranteed.
The approaches [3] and [24] address the challenge of intra-component weaving, however, the authors of [24] do not provide further details on the language, formalism and the used tools.

4.3 Formal approaches for Web Services

Web Services is a typical application domain of component-based development, indeed, in this section we will present a comparative study of formal approaches for the verification of web services. In fact, in tables 4.5, 4.6, 4.7 and 4.8, we have established our comparison according to four dimensions: modeling, verification, evolution and time.

4.3.1 The modeling dimension

This dimension specifies the input language as well as the formal model adopted.

4.3.2 The verification dimension

This dimension gives the formal verification tool used by the approach as well as the details of the tool automating the full approach.

4.3.3 The evolution dimension

This dimension specifies whether the approach addresses the evolution or not, and the properties verified in the case where the evolution is supported. Further, it is also specified whether the approach adopts the aspect-oriented approach as a means of implementing the evolution.

4.3.4 The time dimension

This last dimension indicates whether the approach supports time constraints, this is a real need for several types of systems. Further, the time dimension also specifies whether the verification is performed at design time or dynamically at run time.

4.4 Other approaches

In this section, we will give other approaches having common points with our field of research. Indeed, in tables 4.9, 4.10, 4.11 and 4.12, these approaches are presented and compared according to the same dimensions used during the study of the formal approaches for web services.

4.5 Conclusion

In this chapter, we have presented similar works to our research, the works has been classified mainly into three families: approaches to formalize software components, formal approaches addressing the combination of components and aspects, and formalization of

Table 4.5: Formal Approaches for Web Services - Modeling

Approaches	Input langage	Formalism
JAP'11 [18]	WS-CDL	TA
WSEAS'06 [19]	WS-CDL	TA
UBICOMM'07 [17]	WS-CDL / WS-BPEL	TA
ENTCS'06 [30]	WSCI / WSCDL	TA
CLEI'07 [31]	WSCDL	TA
ENTCS'06 [77]	μ-BPEL	TA
arXiv'11 [47]	none	TA
INFORSID'10 [41]	BPEL/ WSCL	TA
ICWS'09 [40]	(OWL-S)	TA
ARES'06 [50]	BPEL	TA
ICWS'07 [81]	WSBPEL	EVENT CALCULUS
ECOWS'08 [57]	BPEL	TA
SITIS'07 [56]	BPEL	TA(WS-TEFSM)
ICIS'10 [63]	BPEL4WS	WS TA
IJACT'12 [20]	WSDL	Timed Behavior Automata
FMSE'06 [33]	Orc language	TA
WEBIST'14 [78]	UML	TA

Table 4.6: Formal Approaches for Web Services - Verification

Approaches	Model checker	Tool
JAP'11 [18]	UPPAAL	No
WSEAS'06 [19]	UPPAAL	No
UBICOMM'07 [17]	UPPAAL	Yes
ENTCS'06 [30]	UPPAAL	No
CLEI'07 [31]	UPPAAL	(Yes)
ENTCS'06 [77]	UPPAAL	(Yes)
arXiv'11 [47]	UPPAAL	Yes
INFORSID'10 [41]	UPPAAL	(Yes)
ICWS'09 [40]	UPPAAL	(Yes)
ARES'06 [50]	UPPAAL	(Yes)
ICWS'07 [81]	SPIKE	Yes
ECOWS'08 [57]	No	Yes
SITIS'07 [56]	No	Yes
ICIS'10 [63]	UPPAAL	No
IJACT'12 [20]	UPPAAL	No
FMSE'06 [33]	UPPAAL	(Yes)
WEBIST'14 [78]	UPPAAL	Yes

Table 4.7: Formal Approaches for Web Services - Evolution

Approaches	Evolution checking	Checked properties in evolution	Using AOP
JAP'11 [18]	No		No
WSEAS'06 [19]	No		No
UBICOMM'07 [17]	No		No
ENTCS'06 [30]	No		No
CLEI'07 [31]	No		No
ENTCE'06 [77]	No		No
arXiv'11 [47]	No		No
INFORSID'10 [41]	No		No
ICWS'09 [40]	No		No
ARES'06 [50]	No		No
ICWS'07 [81]	No		No
ECOWS'08 [57]	No		No
SITIS'07 [56]	No		No
ICIS'10 [63]	No		No
IJACT'12 [20]	No		No
FMSE'06 [33]	No		No
WEBIST'14 [78]	Yes	Deadlock, Liveness, Reachability	No

Table 4.8: Formal Approaches for Web Services - Time

Approaches	Time	D/R
JAP'11 [18]	Yes	D
WSEAS'06 [19]	Yes	D
UBICOMM'07 [17]	Yes	D
ENTCS'06 [30]	Yes	D
CLEI'07 [31]	Yes	D
ENTCE'06 [77]	Yes	D
arXiv'11 [47]	Yes	D
INFORSID'10 [41]	Yes	D
ICWS'09 [40]	Yes	D
ARES'06 [50]	Yes	D
ICWS'07 [81]	Yes	D
ECOWS'08 [57]	Yes	D
SITIS'07 [56]	Yes	D
ICIS'10 [63]	Yes	D
IJACT'12 [20]	Yes	D
FMSE'06 [33]	Yes	D
WEBIST'14 [78]	Yes	D

Table 4.9: Other approaches - Modeling

Approaches	Input langage	Formalism
SIGSOFT'13 [46]	MechatronicUML	real-time statecharts
ICSE'09 [67]	System at runtime	Pre/post conditions
FTSCS'13 [75]	EAST ADL/CCSL	TA
ECMFA'12 [37]	UML MARTE	Time Petri Nets
CAI'16 [69]	UML MARTE + CCSL	FIACRE + CDL
ICFEM'14 [36]	UML MARTE	TA
SEFM'13 [48]	UML MARTE + CCSL	TA

Table 4.10: Other approaches - Verification

Approaches	Model checker	Tool
SIGSOFT'13 [46]	UPPAAL	No
ICSE'09 [67]	Kermeta	Yes
FTSCS'13 [75]	UPPAAL	No
ECMFA'12 [37]	TINA	Yes
CAI'16 [69]	OBP	Yes
ICFEM'14 [36]	UPPAAL	Yes
SEFM'13 [48]	UPPAAL	(Yes)

Table 4.11: Other approaches - Evolution

Approaches	Evolution checking	Checked properties in evolution	Using AOP
SIGSOFT'13 [46]	Yes	Atomicity, Consistency, Isolation	No
ICSE'09 [67]	Yes	Deadlock, Comparaison of models	Yes
FTSCS'13 [75]	No		No
ECMFA'12 [37]	No		No
CAI'16 [69]	No		No
ICFEM'14 [36]	No		No
SEFM'13 [48]	No		No

Table 4.12: Other approaches - Time

Approaches	Time	D/R
SIGSOFT'13 [46]	Yes	R
ICSE'09 [67]	No	R
FTSCS'13 [75]	Yes	D
ECMFA'12 [37]	Yes	D
CAI'16 [69]	Yes	D
ICFEM'14 [36]	Yes	D
SEFM'13 [48]	Yes	D

components in the context of web services. Further, other similar works having common points with our work were presented.
We can conclude from this chapter, that a methodology of a verification must include the following main steps: an input language (for example: a language specific to the domain), a formalism, a formal verification tool, and a tool automating the full methodology.

Part II

Our approach

Chapter 5

Presentation of the approach

Our approach is a formal support to verify the synergy of component-oriented development and aspect-oriented development. More precisely, we seek to verify some properties during aspect-oriented evolution for component-based systems. First, we will briefly describe features and restrictions concerning our approach, secondly, we give an overview of our approach as well as its main stages, this also includes a reminder on the aspect-oriented principles so that the reader is more autonomous.

5.1 Features and restrictions

Our formal support verifies aspect-oriented evolutions for component-based systems. In fact, the systems subject to this verification must have the following characteristics and restrictions:

- The architecture of the system is composed of classic components and aspects, the latter are considered as components and are called: *aspect components*.

- Our approach supports flat component models as well as hierarchical models, however, the designer must provide the model of the full component-oriented system, in fact, building the full model automatically from the basic software components through a product operation is not supported in the current version of our tools.

- The system to be verified can be designed according to any component model; our approach is general, it is not specific to a given component model.

- The mode of communication between the components of the architecture is synchronous, the asynchronous mode is not supported by our approach.

- In the rest of this part of the thesis, to put it simply, the expressions "software system" or "basic software system" are equivalent to "component-based system" without the aspects *i.e.,* without evolution.

5.2 Overview of our approach

Modern software systems must evolve in response to changes in their environments or in the needs of their users. One of the effective means to complete this evolution is the aspect oriented approach [54], it allows one to make very powerful transformations on a basic program (*e.g.,* object oriented program) via a behavior injection mechanism called *weaving*. This behavior is encapsulated in a module called *aspect*. The aspect mainly contains two elements: an *advice* representing the code to be injected, and the *point cut* that specifies one or several locations of the injection of this code in the basic program. These locations are called *join points*. A join point can be for example a method call. Depending on the type of the advice *i.e.,* before, after and around, the aspect can be woven before, after, or around *a join point* in the base program.

However, this evolution should be channeled through the use of formal methods and tools, to verify for example, that the additional delays introduced by the evolution do not violate some timed properties, or the evolution is not applied in an inappropriate place, this is mostly due to the improper use of wildcards in the point cut expressions.

In our work we focus on the formal design and timed verification of aspect oriented evolution. We propose first a domain specific language (DSL) for describing a basic software system and its possibles evolutions. Second, defining a formal transformation to timed automata and using the UPPAAL tool [11], we support checking the correctness of the evolution. Our approach, automated with our TiVA tool, makes it possible to check different kinds of properties on it, in particular, thoses including time, such as: is some delay for the execution of a given operation (or between two operations) respected? the execution of the system is deadlock free? does the system ends in a correct state? is the order in which some operations execute the correct one? The aspects are applied in appropriate places in the basic system?

(Figure 5.1) presents the main steps of our approach using our TiVA framework.

Further, our framework supports modeling and verification at two levels:

- At the basic system: this makes it possible to verify that some properties are already correct before the system evolves.
- At the evolution: this step allows verifications of the same properties (and possibly others) after the evolution of the system. Further, our framework supports evolutive models based on simple weaving, as well as models based on multiple weaving.

One of the major goals of software engineering is to ascend into levels of abstraction, thus allowing the user to conceal some details from lower levels. In this sense, Domain Specific Languages or even Architecture Description Languages (ADL) offer the means and tools enabling these possibilities. For example, the wright ADL [4], offers a textual specification language allowing to specify a system in an abstract way, thereafter, an automatic translation is carried out to the CSP formalism (Communicating Sequential Processes), thus allowing to perform checks on the FDR tool.
If we draw a parallel with Wright, the DSL that we propose constitutes an abstract means allowing to describe evolutive aspect-oriented systems, in fact, the DSL provides concepts that do not exist on the UPPAAL tool, in particular, the concept of weaving of aspects (single and multiple), otherwise, if the designer describes his system directly in the form of timed automata, he will be bogged down in the management of the weaving on UPPAAL, this is complex and prone to errors. Further, using the our DSL, the designer specifies his system using high level concepts such as: actions, order of actions, time constraints, as

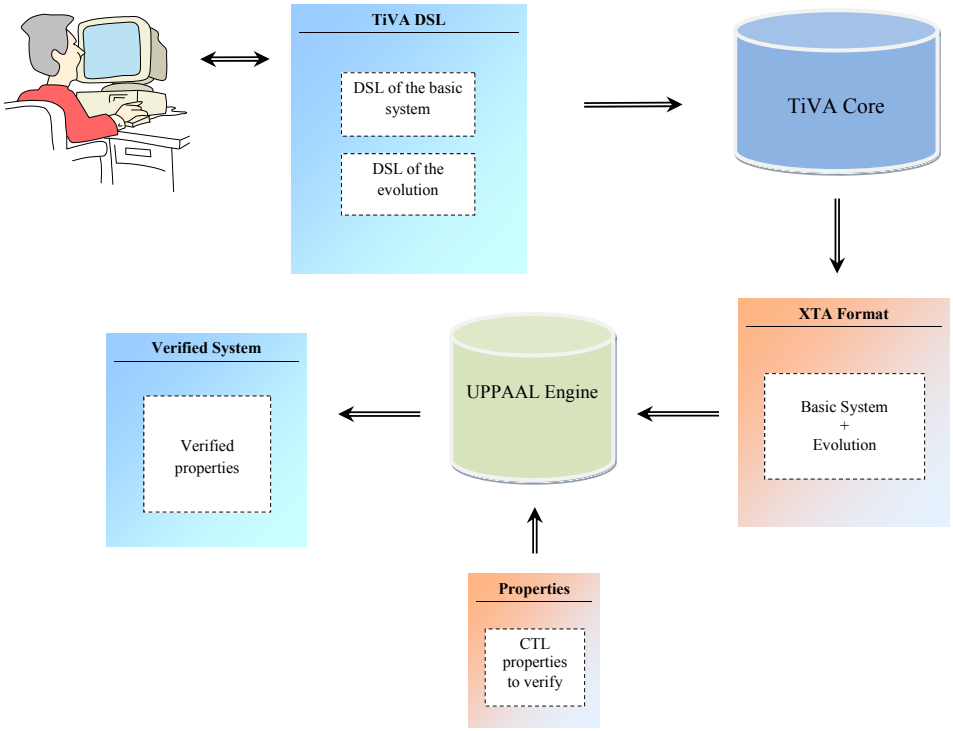

Figure 5.1: TiVA framework

well as weaving operations. If now the designer would like to describe his system directly on the UPPAAL tool, he will be forced to handle lower-level concepts such as: clocks, the calculation of guards, the calculation of invariants, boolean variables, synchronization channels, etc. To compare a description with the DSL and a direct description on UPPAAL, we can give a simple example of weaving an aspect in 30 locations in the base system, this is done on the DSL by a simple expression using a one wildcard, on the other hand, the specification of this weaving directly on the UPPAAL tool requires, among other things, the creation of at least 30 synchronization channels, with the sending and receiving operations, etc. This process is painful and prone to errors. For this reason, similarly to Wright, we created a DSL allowing the designer to make a high level description, then, TiVA Core of our framework makes automatic transformations to a network of timed automata, this allows verification using the UPPAAL model checker which is encapsulated in our TiVA framework.

This part of the thesis is organized as follows. In Chapter 6 we introduce our case study. Chapter 7 presents the models supporting our approach. In Chapter 8, we give the algorithms for the transformation of the models to a network of timed automata. Chapter 9 presents the features related to the verification of properties on the network of timed automata. In Chapter 10, we discuss tool support and evaluation. Chapter 11 compares our work with the state of the art. Finaly, we conclude in Chapter 12.

Chapter 6

Case study

Let us illustrate the use of our approach on a demonstration of *a plane ticket booking system*.

The basic system (Fig. 6.1) is made up of three parts: the (TravelAgency) representing the travel agency, the (BookingPlateform) representing the booking platform, which is used by several travel agencies, and the (BankingSystem).
The agency sends a plan ticket request to the platform, this is accomplished by the (researchSeat) operation. The platform searches in its 30 databases, via (researchBdd) operation, and can respond within 20 to 30 minutes. The platform can give three possible answers by using three possible operations: *available*, *no available*, or *seat found but not still confirmed*, if the person who made the booking does not pay within 120 minutes, this seat will be available. If the seat is available, the platform gives the agency a confirmation period of 120 minutes. If the platform receives the confirmation, a payment request is sent to the bank via the (askPay) operation, this takes between 10 and 15 minutes. If the credit in the bank account is insufficient, the booking procedure is canceled. If the credit is sufficient the payment is made. Afterwards, the platform gives the agency a free cancellation period of 30 minutes. If the agency cancels within this period, the booking procedure is canceled. Exceeding this deadline, the booking is definitively confirmed and the cancellation will be charged.

We want to verify the following properties for this system: P1: (*The partial delay*) The delay (min and max) to have an answer on the availability of a seat, this property allows the travel agency to see if this delay is acceptable. P2: (*The full delay*) The overall time (min and max) of the full transaction, from request step until the payment. P3: (*The consistency of the system*) The system has no deadlock. P4: (*Correct termination*) System execution always ends with cancellation or final confirmation.

First, the designer checks these properties on his basic system *i.e.,* before evolution. Thereafter, the designer will need to evolve his system by inserting new parts following the aspect-oriented approach. Indeed, the designer should add two additional parts *i.e.,* two aspects. After the execution of the (researchBdd) operation, the platform will execute an (externalResearch) aspect allowing to make an additional research in other platforms, this takes a delay between 35 and 40 minutes. In the case that the environment is insecure, an (Encryption) aspect ensuring a more sophisticated encryption system is added, before any sending operation to the bank.

After the evolution of the system, the designer has to check the properties again, to

see if the delays are still acceptables or not. Further, after the evolution, the designer can also checks other properties (in addition to the four mentioned above), for example: P5: (*Ordering*) System always performs internal researches before performing external researches.

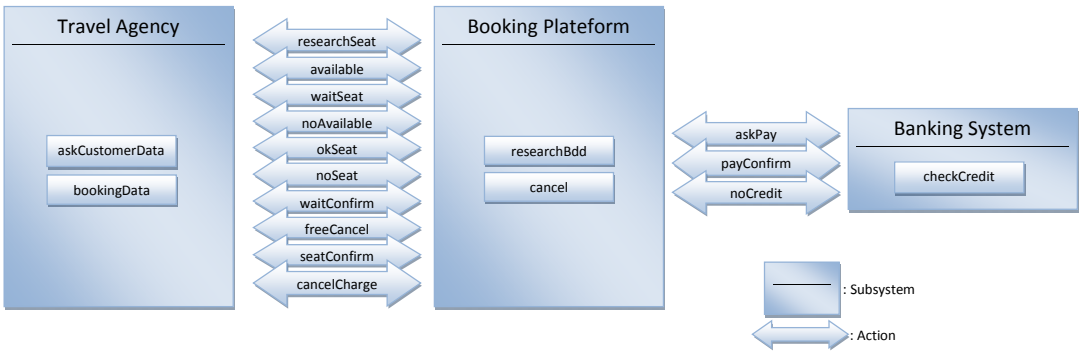

Figure 6.1: Booking system.

Chapter 7

Modeling

In this chapter, we present the models that support, in our framework, the design and verification of a basic software system and its aspect-oriented evolution. Our models are defined in terms of signatures and behavioral descriptions. Our models are realized into a domain specific language that is exemplified all along with the formal definitions (the global model for our case study in our DSL being given in A). Let's remember that in the next chapters, a basic software system (or a software system) means a component-based system without aspect-oriented evolution.

7.1 Signatures

The signature part of a software system describes the purely functional informations about this system, *i.e.,* the set of the operations in the software system.

Definition 1 (Signature)**.** Being given a set $\mathcal{B}$ of basic systems, a set $\mathcal{O}$ of operations, and an element b in $\mathcal{B}$, a signature for b is $Sig_b = \mathcal{O}$

When clear from the context, a signature Sig_b can be simply denoted by Sig. An operation can be: a required operation *i.e.,* a service required by the component, a provided operation *i.e.,* a service provided by the component, or an internal operation of a component.

Example 1 (Signature of the Booking basic system)**.** The signature of the Booking system (presented in our case study chapter 6).

```
1 signature
2 researchSeat , researchBdd , noAvailable , waitSeat , noSeat ,
3 available , okSeat , waitConfirm , cancel , askPay , noCredit ,
4 payConfirm , freeCancel , seatConfirm , CancelCharg ;
```

7.2 Behavioral descriptions

Signatures specify systems in terms of the operations. However, the order in which these operations take place is not given in signatures. This is the purpose of a another specification layer, a behavioral one [12]. In the literature, different models can be found to play this role, *e.g.,* Labelled Transition Systems (LTS) [93], Petri Nets [64, 10], Process Algebras [4], or Timed Automata (TA) [6]. We chose to rely on Labelled Transition Systems, due to their simplicity, and their extensibility.

Given that the verification step is done on TA, we could have considered using TA directly. However, with LTS we have a high level modeling, while with TA the modeling is more complex and errors prone, since, by definition, a TA is more complex than an LTS [1]. In our model, the labels correspond to the different operations of the software system.

Definition 2 (Behavior)**.** Being given a signature Sig_b for a system b, a behavior for b with reference to Sig_b is a tuple $Beh_b = (A, S, s_0, F, T)$ where:

- $A = A^t \cup A^{nt}$ is the alphabet, where
  - $A^t = \{o\} \times \mathbb{N}^+ \times \mathbb{N}^+$ such that $o \in \mathcal{O}$, is the alphabet of timed operations
  - $A^{nt} = \{o\}$ such that $o \in \mathcal{O}$, is the alphabet of simple operations
- S is a finite and non empty set of states,
- $s_0 \in S$ is the initial state,
- $F \subseteq S$ is a non empty set of final states, and
- $T \subseteq S \times A \times S$ is the set of transitions[2].

When clear from the context, a behavior Beh_b can be simply denoted by Beh. The initial state corresponds to the state the system starts its execution. The final states correspond to correct end-of-service states for the system. This is used, *e.g.,* to make a difference between a blocked system and one that has terminated. An element (s_1, a, s_2) in T can be denoted as $s_1 \xrightarrow{a} s_2$. Operations can be simples *i.e.,* without time constraints, or timed operations that last over time between a minimum delay x and a maximum delay y as in Figure 7.1, *e.g.,* $(b, 3, 4)$ denotes an operation that lasts between 3 and 4 units of time (uot). Note that time is expressed in terms of units of time, which the designer can then assign for example to seconds or minutes, depending on the system to be modeled.

With reference to Definition 2, we require a consistency constraint:

- Correct time intervals: $\forall (a, x, y) \in A^t,\ x \leqslant y$

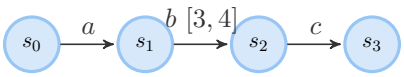

Figure 7.1: The first and the last transitions denote two operations without a time constraints. The second one denotes an operation that takes between 3 and 4 uot.

Example 2 (Behavior of the Booking basic system)**.** The behavior of the Booking system (presented in our case study chapter 6) is given in Figure 7.2.

```
behavior
init s0;
final s3,s5,s8,s10,s13,s14,s15;
trans
s0:researchSeat:s1,
s1:researchBdd[20-30]:s2,
```

[1] The semantics of a TA in [11] is defined on an LTS. Further, a TA can be seen as a more elaborate LTS, with clocks and other concepts.

[2] Formally, this set can be empty, even if this is not usually the case in practice.

```
s2:noAvailable:s3,
s2:waitSeat[120-120]:s4,
s4:noSeat:s5,
s2:available:s6,
s4:okSeat:s6,
s6:waitConfirm[120-120]:s7,
s7:cancel:s8,
s7:askPay[10-15]:s9,
s9:noCredit:s10,
s9:payConfirm:s11,
s11:freeCancel[0-30]:s12,
s12:cancel:s13,
s12:seatConfirm[1-1]:s14,
s14:CancelCharg:s15;
end
```

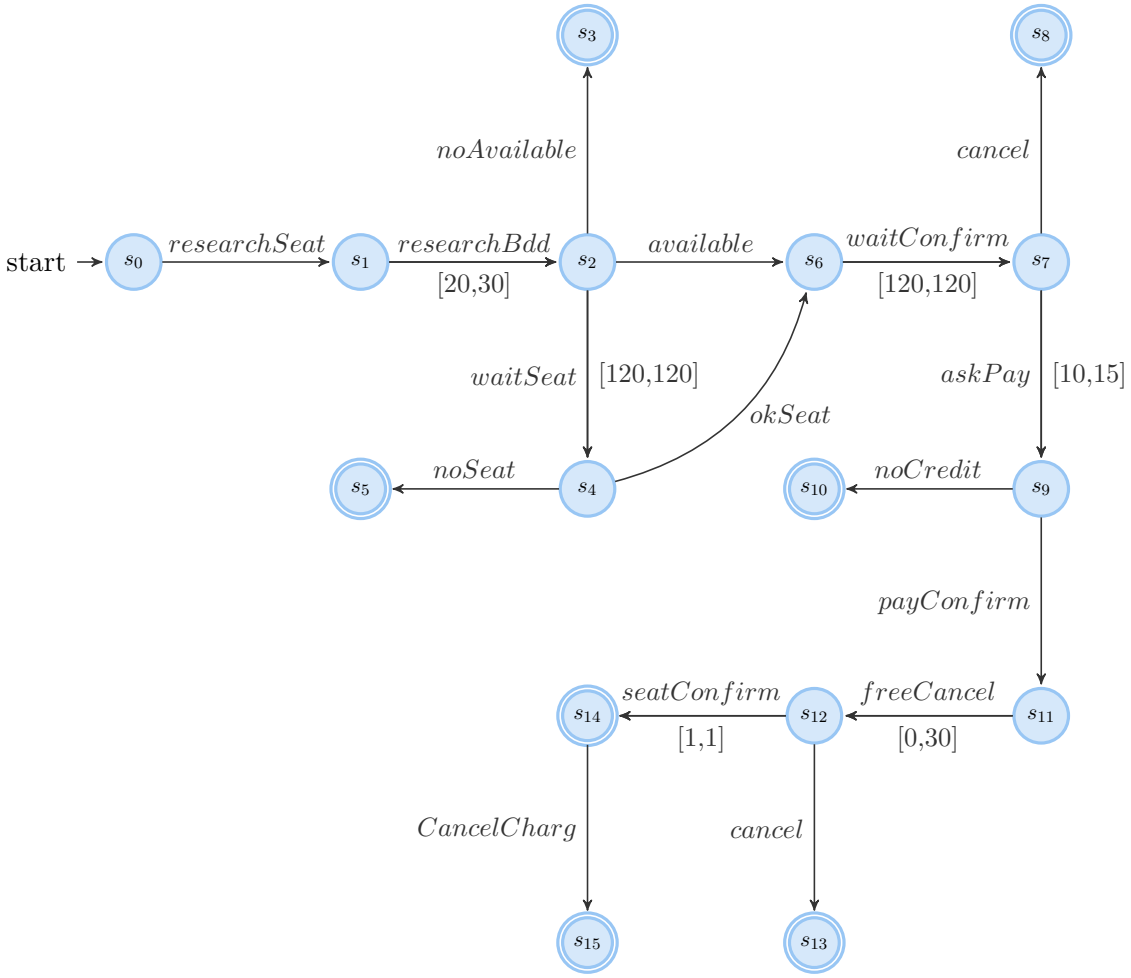

Figure 7.2: Model of the basic system Booking.

7.3 Basic software system

A basic software system is described with two levels: a signature and a behavior.

Definition 3 (Basic Software system)**.** A basic software system is a couple $b = (Sig_b, Beh_b)$ where:

- Sig_b is the signature of the software system, and

- Beh_b is the behavior of the software system, defined with reference to Sig_b.

Example 3 (Booking basic software system)**.** The definition of the Booking basic software includes the signature and the behavior given above.

```
system Booking;

signature
researchSeat, researchBdd, noAvailable, waitSeat, noSeat,
available, okSeat, waitConfirm, cancel, askPay, noCredit,
payConfirm, freeCancel, seatConfirm, CancelCharg;

behavior
init s0;
final s3, s5, s8, s10, s13, s14, s15;
trans
s0:researchSeat:s1,
s1:researchBdd[20-30]:s2,
s2:noAvailable:s3,
s2:waitSeat[120-120]:s4,
s4:noSeat:s5,
s2:available:s6,
s4:okSeat:s6,
s6:waitConfirm[120-120]:s7,
s7:cancel:s8,
s7:askPay[10-15]:s9,
s9:noCredit:s10,
s9:payConfirm:s11,
s11:freeCancel[0-30]:s12,
s12:cancel:s13,
s12:seatConfirm[1-1]:s14,
s14:CancelCharg:s15;
end
```

7.4 Evolution

7.4.1 Aspects, Join Points, and Weaving

In our approach, evolution is achieved using an aspect-oriented mechanism: *weaving*. Weaving consists in interrupting the execution of the system at some point, called *join point*, executing a specific behavior of interest, called *aspect*, and then resuming the standard system execution where it stopped. This can have an effect on the properties of the system, either behavioral ones (deadlocks could be introduced) or time-related ones (the execution of the overall system or subparts of it may get out of specified time bounds). Further, aspects may be applied to inadequate operations, this is due to the use of wildcards in the expressions to specify join points. Aspects will correspond to specific kinds of software system to add, join points are specific operations in the basic system, and weaving is a composition process of an aspect with one or several operations of the basic software system.

Definition 4 (Aspect)**.** Being given a set $\mathcal{B}$ of basic software systems, an aspect is a tuple $\alpha = (b, Trigger, Stop)$ where

- $b \in \mathcal{B}$, and
- $Trigger, Stop \in A^{nt}(Beh_b)$.

An aspect can be defined over a basic software system. Further, we require that it has two distinguished operations, $Trigger$ and $Stop$, respectively specifying where the aspect starts and terminates its execution. In the aspect-oriented approach, weaving can

be achieved following three kinds of *advices*: before, after, and around. In this work we support the first two. A weaving with a before advice will take place before executing an operation in the basic software system. A weaving with an after advice will take place after executing an operation in the basic software system. A weaving is given as the aspect, the operation in the basic software system, and the advice. We use $isTrigger(o)$ to check if an operation o is the specific $Trigger$ operation for the aspect. We use $isStop(o)$ to check if an operation o is the specific $Stop$ operation for the aspect. Note that a formal definition of a weaving will be given later.

Example 4 (Example of an aspect)**.** The DSL representing the Encryption aspect is given below.

```
1
2  aspect Encryption;
3
4  signature
5  inputData, encryptData, outputRes;
6
7  behavior
8  init s0;
9  final s3;
10 trans
11 s0:inputData:s1:trigger,
12 s1:encryptData[13-18]:s2,
13 s2:outputRes:s3:stop;
14 end
```

Definition 5 (Weaving)**.** Being given a set $\mathcal{B}$ of software systems, and a set A of aspects, a weaving is a tuple $\omega = (b, j, \alpha, ad)$ where:

- $b \in \mathcal{B}$ is the basic system without evolution on which the aspects will be woven,
- $j \in A(Beh_b)$ is the operation in the basic system, on which the weaving aspect will be woven,
- $\alpha \in A$ is an aspect, and
- $ad \in \{before, after\}$, is an advice.

Example 5 (Some aspects and weavings)**.** In our case study, when the environment is insecure, an aspect Encryption is woven before the execution of the operation of sending data to the banking system. We can also find the aspect externalResearch allowing to search for seats in other platforms, this aspect is woven after the researchBdd operation.

```
1
2  Weaving (Booking:askPay[10-15]:Encryption:before);
```

7.4.2 Adapters

Several aspects can operate at a given join point. The role of aspect adapters (or adapters for short) is to specify possible constraints on the application of aspects in such a case, and more generally between aspects that are to be applied on a software system. We define two different kinds of adapters: precedence and mutex. Let us note that the possible constraints that can exist between the aspects are summarized in [80] and [85].

Definition 6 (Aspect Adapter)**.** An (aspect) adapter is a tuple $\gamma = (\omega_1, \omega_2, kind)$ where ω_1 and ω_2 are two weavings and $kind$ is an element in $\{prec, mutex\}$. Note that, two

aspects concerned by the same join point but having two different advices are considered to be independent and will not be bound by the same adapter since they are woven at two different locations on this same joint point, an aspect with an advice before is woven on the call of an operation, while an aspect with an advice after is woven after the execution of an operation.

An adapter $\gamma = (\omega_1, \omega_2, kind)$ can be denoted by $\boxed{kind}\ \{\omega_1, \omega_2\}$.

Precedence adapter

The precedence adapter is used to specify an ordering of aspects, as found in several aspect oriented languages, *e.g.,* in AspectJ [52]. A $\boxed{prec}\ \{n_1, n_2\}$ adapter specifies that at the join point of interest the n_1 aspect should be applied before the n_2 aspect.

Example 6 (Precedence adapter)**.** In our case study, in the previous example, we have an aspect externalResearch to search in other platforms, some neighboring platforms use another data format, in this case the aspect externalResearch must be followed by the execution of the aspect formatConversion, and these two aspects must be woven in this order after the same operation.

```
1 Adapter (Booking:researchBdd[20-30]:externalResearch:formatConversion:after:prec);
```

Mutual exclusion adapter

The mutual exclusion adapter is used when two aspect instances should not be applied at the same time, *e.g.,* because they have identical or contradictory purposes. The choice between one and the other depends on the execution context.

Example 7 (Mutex adapter)**.** In our case study, when two transmission media are used with the banking system (wireless and intranet), each medium requires a specific encryption system, in this case, we must use two aspects for encryption: encryptionWifi and encryptionLan. Naturally, these two aspects are used in mutual exclusion.

```
1 Adapter(Booking:askPay[10-15]:encryptionLan:encryptionWifi:before:mutex);
```

7.4.3 Aspect Oriented System

Definition 7 (Aspect Oriented System)**.** Being given a set $\mathcal{B}$ of basic systems, and an element b in $\mathcal{B}$, an aspect oriented system is a tuple $\mathcal{AOS} = (b, A, W, \Gamma)$ With:

- A is a set of aspects,
- W is a set of weavings ω defined w.r.t. b and A, and
- Γ is a set of aspect adapters γ defined w.r.t. W.

Example 8 (Aspect Oriented system of a booking system)**.** The full DSL of our case study is in A.

```
1
2 system Booking;
3
4 signature
5 researchSeat,researchBdd,noAvailable,waitSeat,noSeat,
6 available,okSeat,waitConfirm,cancel,askPay,noCredit,
```

```
payConfirm,freeCancel,seatConfirm,CancelCharg;

behavior
init s0;
final s3,s5,s8,s10,s13,s14,s15;
trans
s0:researchSeat:s1,
s1:researchBdd[20-30]:s2,
s2:noAvailable:s3,
s2:waitSeat[120-120]:s4,
s4:noSeat:s5,
s2:available:s6,
s4:okSeat:s6,
s6:waitConfirm[120-120]:s7,
s7:cancel:s8,
s7:askPay[10-15]:s9,
s9:noCredit:s10,
s9:payConfirm:s11,
s11:freeCancel[0-30]:s12,
s12:cancel:s13,
s12:seatConfirm[1-1]:s14,
s14:CancelCharg:s15;
end

aspect externalResearch;

signature
inputData, searchData, outputRes;

behavior
init s0;
final s3;
trans
s0:inputData:s1:trigger,
s1:searchData[10-15]:s2,
s2:outputRes:s3:stop;
end

Weaving (Booking:researchBdd[20-30]:externalResearch:after);
```

Chapter 8

Model transformation

In order to support the verification of software systems that can be modeled using our approach, we give them a formal semantics by transforming them into (networks of) timed automata. This paves the way, in a second step to perform verification using UPPAAL [58]. We begin by recalling the formal definition of a timed automata, and then we explain how to take into account basic software system as well as its evolution in the formal transformation of a model into timed automata.

8.1 Timed Automata

Timed automata are an extension of automata with state invariants, transition conditions, and transition actions that relate to a set of clocks. They make great use of clock constraints that are conjunctions of atoms of the form $cl_1 \bowtie n$ or $cl_1 - cl_2 \bowtie n$, where cl_1 and cl_2 are clocks, $\bowtie$ is an operator in $\{<, \leqslant, =, \geqslant, >\}$, and n is a natural number [18]. Given a set of clocks C, the set of clock constraints built over C is denoted by $B(C)$.

Definition 8 (Timed Automaton, from [18])**.** A timed automaton is a tuple (L, l_0, C, A, E, I, U), where:

- L is a set of locations,
- $l_0 \in L$ is the initial location,
- C is the set of clocks,
- A is a set of actions, co-actions[1], and the internal τ-action,
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action (taken in A), a guard (a clock constraint in $B(C)$ and a set of clocks to be reset, and
- $I : L \to B(C)$ assigns invariants to locations.
- $U \subseteq L$ is the subset of urgent locations.

Guards represent the conditions that enable a transition when they are satisfied. Invariants are conditions associated to locations, that specify that the system can stay in the location if and only if the invariant is satisfiable. An element (l_1, a, g, R, l_2) in E can be denoted as $l_1 \xrightarrow{[g]a/R} l_2$. When g or R are empty, they can be omitted. In figures a

[1]Typically, an action is an emission on a communication channel, and the corresponding co-action is the reception of it.

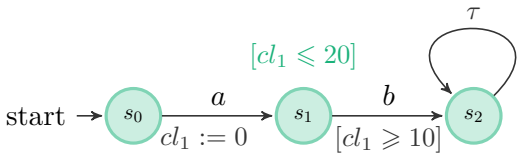

Figure 8.1: A simple example of a timed automaton.

clock reset for a clock cl is denoted as $cl := 0$, and both guards and invariants are denoted as $[g]$. However, guards are associated to edges and invariants to locations (see Fig. 8.1). Further, urgent locations freeze time; *i.e.,* time is not allowed to pass when a process is in an urgent location. Semantically, urgent locations are equivalent to: adding an extra clock x that is reset on every incoming edge, and adding an invariant $x <= 0$ to the location.

Example 9 (Timed Automata)**.** Figure 8.1 shows an example of a timed automaton. A clock cl_1 is reset when action a is executed. The invariant on state s_1 means that it is possible to stay in this state for up to 20 units of time. The guard on the edge labelled with b means that at least 10 units of time must have passed after entering in s_1 before possibly doing b. Altogether, this expresses that b has to occur between 10 and 20 units of time after a.

8.2 Transformation of Aspect-Oriented Systems

In this section, we give the algorithms for the transformation from our model to a network of timed automata.

8.2.1 Basic system to timed automaton

Program 1 transforms a basic software system to a timed automaton using the function *LTStoTA*. Figure 8.2 shows the transformation of the LTS for the booking system (presented in Figure 7.2) to a timed automaton.

8.2.2 Aspects to timed automata

Program 2 transforms an aspect to a timed automaton using the function *AStoTA*. First, this function uses the function *LTStoTA* to build a timed automaton. Thereafter, synchronization channels are inserted.

8.2.3 Weaving

Programs 3 and 4 describe the weaving process using the function *Weaving*. First, this function uses the function *LTStoTA* to build a timed automaton for the basic system. Second, function *AStoTA* is called to build the timed automata for the aspects. Finally, synchronization channels are inserted.

8.2.4 Correctness of transformation

The idea of the translation is to translate the base element in DSL, which is an LTS, to the base element in the model in timed automata, which is a timed automaton. This is done by Program 1. Then, once we have the basic timed automata, Programs 2, 3 and 4, compose the basic timed automata to obtain a global model in the form of a network

Program 1 Transformation of a basic system to a timed automaton.

-- retrieves a timed automaton
-- from a basic system
input : $b,\ with\ Beh_b = (A^{LTS}, S, s_0, F, T),\ a\ basic\ software\ system$
output : $TA = (L, l_0, C, A^{TA}, E, I, U)$
-- 1. base
$L = S,\ C = \emptyset,\ l_0 = s_0,$
$A^{TA} = \mathcal{O} \cup \{tau\},\ E = \emptyset,\ I = \emptyset,$
$U = S$
-- 2. Final States
for each s_i^F *in* F
create a new state $s \notin L$
$L = L \cup \{s\}$
$E = E \cup \{s_i^F \xrightarrow{tau} s\ ,\ s \xrightarrow{tau} s\}$
end for
-- 3. Transitions without time constraints
for each $s_1 \in S$
for each $s_1 \xrightarrow{a} s_2 \in T$ *such that* $a \in A^{nt}$
$E = E \cup \{s_1 \xrightarrow{tau} s_2\}$
end for
end for
-- 4. Transitions with time constraints
for each $s_1 \in S$
for each $s_1 \xrightarrow{a\ x\ y} s_2 \in T$ *such that* $a \in A^t$
create a new state $s \notin L$
create a new clock cl
$L = L \cup \{s\}$
$C = C \cup \{cl\}$
$I(s) = I(s) \wedge cl \leqslant y$
$E = E \cup \{s_1 \xrightarrow{tau/cl:=0} s, s \xrightarrow{[cl \geqslant x]\ a} s_2\}$
end for
end for
-- end
return $(L, l_0, C, A^{TA}, E, I, U)$

Program 2 Transformation of an aspect to a timed automaton.

-- retrieves a timed automaton
-- from an aspect
input : $asp = (b, Trigger, Stop)$ *with* $Beh_b = (A^{LTS}, S, s_0, F, T)$ *asp is an aspect. ch is a synchronization chanal.*
output : $TA = (L, l_0, C, A^{TA}, E, I, U)$, *a timed automaton*
-- 1. base generation
$TA = \boxed{(LTStoTA(b)}$
-- 2. Chanals generation
for each $s_1 \xrightarrow{a} s_2 \in T$ *such that* $isTrigger(a)$ *or* $isStop(a)$
  if $isTrigger(a)$
    For $s_1 \xrightarrow{tau} s_2 \in E$
      $E = E - \{s_1 \xrightarrow{tau} s_2\}$
      $E = E \cup \{s_1 \xrightarrow{ch?} s_2\}$
      $U = U - \{s_1\}$
    end for
  else
    For $s_1 \xrightarrow{tau} s_2 \in E$
      $E = E - \{s_1 \xrightarrow{tau} s_2\}$
      $E = E \cup \{s_1 \xrightarrow{ch!} s_2\}$
    end for
  end if
end for each
return $(L, l_0, C, A^{TA}, E, I, U)$

Program 3 Weaving. Part 1 (before advice).

-- retrieves a network of timed automata
-- from a basic system and aspects
input : $weaving = (b, j, aspect, ad)$, *a weaving*
output : $TA_b = (L, l_0, C, A^{TA}, E, I, U)$ $TA_{aspect\ (i)} = (L, l_0, C, A^{TA}, E, I, U)$
$i = 1..n$, *n is the number of aspect timed automata*
$TA_b = \boxed{LTStoTA(b)}$
if $ad = before$
if $j \in A^{nt}(Beh_b)$
for each $s_1 \xrightarrow{j} s_2 \in T_b$
$E_{TA_b} = E_{TA_b} - \{s_1 \xrightarrow{tau} s_2\}$
create a new chanal ch
create a new state $s_B \notin L_{TA_b}$
$L_{TA_b} = L_{TA_b} \cup \{s_B\}$
$E_{TA_b} = E_{TA_b} \cup \{s_1 \xrightarrow{ch!} s_B,\ s_B \xrightarrow{ch?\ /tau} s_2\}$
$TA_{aspect_i} = \boxed{AStoTA(aspect, ch)}$
end for each
end if
if $j \in A^{t}(Beh_b)$
for each $s_1 \xrightarrow{j} s_2 \in T_b$
for $s_1 \xrightarrow{cl:=0} s_A$ *and* $s_A \xrightarrow{[cl>=x]} s_2 \in E_{TA_b}$
$E_{TA_b} = E_{TA_b} - \{s_1 \xrightarrow{cl:=0} s_A\}$
create a new chanal ch
create a new state $s_B \notin L_{TA_b}$
$L_{TA_b} = L_{TA_b} \cup \{s_B\}$
$E_{TA_b} = E_{TA_b} \cup \{s_1 \xrightarrow{ch!} s_B,\ s_B \xrightarrow{ch?\ /cl:=0} s_A\}$
$TA_{aspect_i} = \boxed{AStoTA(aspect, ch)}$
end for
end for each
end if

Program 4 Weaving. Part 2 (after advice).

$\underline{if}\ ad = after$
  $\underline{if}\ j \in A^{nt}(Beh_b)$
    $\underline{for\ each}\ s_1 \xrightarrow{j} s_2 \in T_b$
      $E_{TA_b} = E_{TA_b} - \{s_1 \xrightarrow{tau} s_2\}$
      create a new chanal ch
      create a new state $s_B \notin L_{TA_b}$
      $L_{TA_b} = L_{TA_b} \cup \{s_B\}$
      $E_{TA_b} = E_{TA_b} \cup \{s_1 \xrightarrow{tau\ /ch!} s_B,\ s_B \xrightarrow{ch?} s_2\}$
      $TA_{aspect_i} = \boxed{AStoTA(aspect, ch)}$
    $\underline{end\ for\ each}$
  $\underline{end\ if}$
  $\underline{if}\ j \in A^{t}(Beh_b)$
    $\underline{for\ each}\ s_1 \xrightarrow{j} s_2 \in T_b$
      $\underline{for}\ s_1 \xrightarrow{cl:=0} s_A$ *and* $s_A \xrightarrow{[cl>=x]} s_2 \in E_{TA_b}$
      $E_{TA_b} = E_{TA_b} - \{s_A \xrightarrow{[cl>=x]} s_2\}$
      create a new chanal ch
      create a new state $s_B \notin L_{TA_b}$
      $L_{TA_b} = L_{TA_b} \cup \{s_B\}$
      $E_{TA_b} = E_{TA_b} \cup \{s_A \xrightarrow{[cl>=x]\ ch!} s_B,\ s_B \xrightarrow{ch?} s_2\}$
      $TA_{aspect_i} = \boxed{AStoTA(aspect, ch)}$
      $\underline{end\ for}$
    $\underline{end\ for\ each}$
  $\underline{end\ if}$
$\underline{end\ if}$
$\underline{return\ TA_b,\ TA_{aspect\ (i)}}$

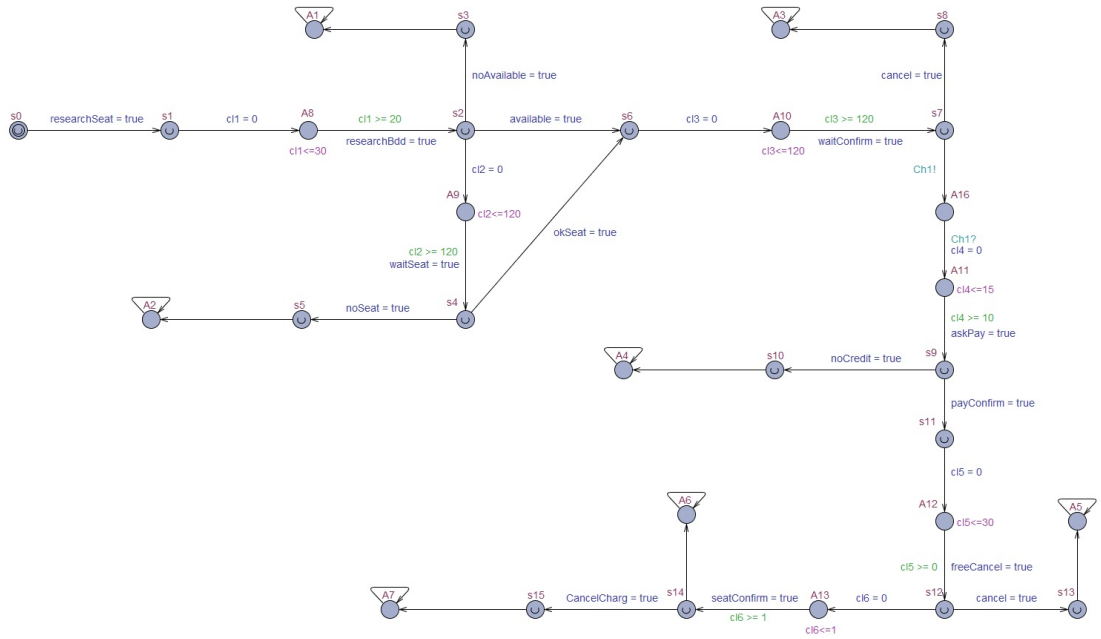

Figure 8.2: Timed automaton for Booking system design

of timed automata, representing an aspect oriented system. Therefore, the proof of the correctness of the translation consists in proving the correctness of the basic transformation *i.e.,* Program 1.
In Program 1, the elements 1,2 and 3 represent a simple cloning of an LTS to a TA (the locations of a TA are the states of an LTS, the initial location is the initial state, etc), indeed, the translation is specified by element 4 of Program 1. Element 4 *i.e.,* transitions with time constraints, (in Program 1) specifies the translation of a transition decorated by an action a with a minimum delay x and a maximum delay y (on an LTS). In fact, we have to provide proof that this same action a is executed between units of time x and y on the corresponding TA.
In what follows, we will first present the semantics of a timed automaton, thereafter we will provide the proof of correctness of the translation.

Semantics of timed automaton

In this section, we will present the semantics of a timed automaton, this will be the basis of the proof of correctness of the transformation.

Definition 9 (Semantics of TA, from [18])**.** Let (L, l_0, C, A, E, I) be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \rightarrow \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times (\mathbb{R}_{\geqslant 0} \cup A) \times S$ is the transition relation such that:

- *Rule* 1 : $(l, u) \xrightarrow{d} (l, u + d)$ if $\forall d' : 0 \leqslant d' \leqslant d \Rightarrow u + d' \in I(l)$, and
- *Rule* 2 : $(l, u) \xrightarrow{a} (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ s.t. $u \in g$, $u' = [r \mapsto 0]u$, and $u' \in I(l')$,

where for $d \in \mathbb{R}_{\geqslant 0}$, $u + d$ maps each clock cl in C to the value $u(cl) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \backslash r$.

Proof of correctness of transformation

Here we will prove that an action a (defined on an LTS) with a minimum execution time x and a maximum execution time y, will be executed between the time units x and y on the corresponding TA, after its transformation by Program 1.

Proof. Let a_{LTS} be an action a defined on an LTS s.t. (s_1, a, s_2) with (a, x, y) (see section 7.2)

The translation of the action a is: $Program\ 1(a_{LTS}) = a_{TA}$ s.t. a_{TA} denotes the corresponding TA of action a after the translation of a.

$a_{TA} = (L = L \cup \{s\}, l_0, C = C \cup \{cl\}, A, E = E \cup \{s_1 \xrightarrow{tau/cl:=0} s, s \xrightarrow{[cl \geqslant x]\ a} s_2\}, I(s) = I(s) \wedge cl \leqslant y)$.

We use $exec(a)$ to check if an action a is executed, a is executed when $exec(a) = true$.

For $cl = 0$ (The automaton is in state (s_1, cl)):

- w.r.t *Rule* 1: given that $cl + d' \notin I(s_1)$ because s_1 is an urgent location, then conditions are not verified for *Rule* 1.
- w.r.t *Rule* 2: given that $cl \in g$, then conditions are verified for *Rule* 2 $\Rightarrow$ The new state of the automaton is (s, cl) with $cl = 0$.

For $0 \leqslant cl < x$: (The current state is (s, cl)):

- w.r.t *Rule* 1: given that $cl < x$ and $x \leqslant y$ then $cl < y$ that means $cl \in I(s)$, then conditions are verified for *Rule* 1 $\Rightarrow$ The new state of the automaton is (s, cl) with $cl < x$.
- w.r.t *Rule* 2: given that $cl < x$ that means $cl \notin g$, then conditions are not verified for *Rule* 2.

Conclusion: For $0 \leqslant cl < x$: the new state is (s, cl) with $0 \leqslant cl < x$.

For $x \leqslant cl \leqslant y$: (The current state is (s, cl)):

- w.r.t *Rule* 1: given that $cl \leqslant y$ that means $cl \in I(s)$, then conditions are verified for *Rule* 1 $\Rightarrow$ The new state of the automaton is (s, cl) with $x \leqslant cl \leqslant y$.
- w.r.t *Rule* 2: given that $cl \geqslant x$ that means $cl \in g$, then conditions are verified for *Rule* 2 $\Rightarrow$ The new state of the automaton is (s_2, cl) with $cl = 0$ and $exec(a) = true$.

Conclusion: For $x \leqslant cl \leqslant y$: the two possible states are: (s, cl) with $x \leqslant cl \leqslant y$ or (s_2, cl) with $cl = 0$ and $exec(a) = true$.

For $cl > y$: (The current state is (s_2, cl)):

- w.r.t *Rule* 1: given that s_2 is an urgent location that means $I(s_2) = \{0\}$, then conditions are not verified for *Rule* 1.
- w.r.t *Rule* 2: suppose that conditions for *Rule* 2 are verified, then the new state $s' \notin \{s_1, s_2, s\}$ and $exec(a) = false$ given that for (s_2, b, g, r, s'), $b \neq a$.

General conclusion: $exec(a) = true$ if and only if $x \leqslant cl \leqslant y$.

□

Chapter 9

Verification and simulation

Once the model of our system is automatically transformed into a network of timed automata, one can do verifications and simulations using the UPPAAL model checker.

9.1 Verification

We present here some examples of typical use of verification.

9.1.1 Deadlock free

The system is deadlock free. This is done using the following property:

$$A[]\ not\ deadlock$$

9.1.2 Correct termination

The system always ends its execution in a correct state. Here we verify that for all possible executions the system can always reach one of the final states. Note that the use of this property requires knowing, depending on the context, the meaning of a correct state, in other words, the meaning of each final state used in the property. For instance, the final states used in this type of properties can be: acceptance states and error states, only acceptance states, or only error states. For the latter case, for example, the verification consists in detecting whether the system is still terminating its execution in an error state.This is done using the following property:

$$A <>\ \phi\ or\ \psi$$

For example:

$$A <>\ booking.final_1\ or\ booking.final_2\ or\ ...\ booking.final_n$$

9.1.3 Ordering

An operation $o1$ is always executed before an operation $o2$, this is done using this property:

$$A[]\ (\phi\ imply\ \psi)$$

For example, in our booking system, we want to check that a (researchBdd) operation is always executed before the (searchData) operation, to check if the system always starts by searching locally.

$$A[]\ (externalResearch.searchData == true\ imply\ booking.researchBdd == true)$$

Such that, (researchBdd) and (searchData) are boolean variables initialized to false. This property verifies that always when (searchData) is true (researchBdd) has already been set to true.
Let us note that TiVA Core generates a boolean variable for each operation, having the name of this operation. In fact, this boolean variable can be used to write CTL properties and verify them.

9.1.4 The delay of execution

This property is used to check an execution time such as, a delay of execution of an operation, a delay between the execution of two operations, or the overall delay of execution of a system. More generally, this property verifies the delay between two instants of time in the system t_1 and t_2, for this, the global clock X is used *i.e.,* an observer, this clock must be reset at instant t_1, then the property consists in comparing the value of the clock with a given value at instant t_2. The general form of this property is as follows:

$$A\ []\ \phi\ imply\ clock \leqslant n$$

In our case study, we can give as an example the verification of the overall delay of the full booking process, this is verified by the following property:

$$A\ []\ booking.seatConfirm\ imply\ X \leqslant 315$$

Which translates to: for all possible executions, when the system is in the (seatConfirm) operation, the clock X is lower or equal to 315 minutes.

9.2 Simulation

Our framework allows the designer to check properties without using an external tool, since the uppaal engine is encapsulated in TiVA framework. However, to have counter examples for unverified properties or to make some specific verifications, for example inspecting places in the basic program where the aspects are woven, the designer has to do simulations with an external tool *i.e.,* UPPAAL tool, using the model of network of timed automata produced by TiVA framework.

Chapter 10

Tool Support and Evaluation

10.1 Tool support

Our work is part of the TiVA project [1] that enables one to design correct timed software systems with their aspect-oriented evolution, using an expressive domain specific modeling language. In order to model and enable the verification of models, we have developed a TiVA Framework, it includes two tools that we have developed separately: TiVA DSL and TiVA Core.

10.1.1 TiVA DSL

TiVA DSL (Figure 10.1) is a tool (an Eclipse plugin) that enables to edit, check syntax, and save the DSL model (tiva file) to be verified (see A for the full DSL of our case study).

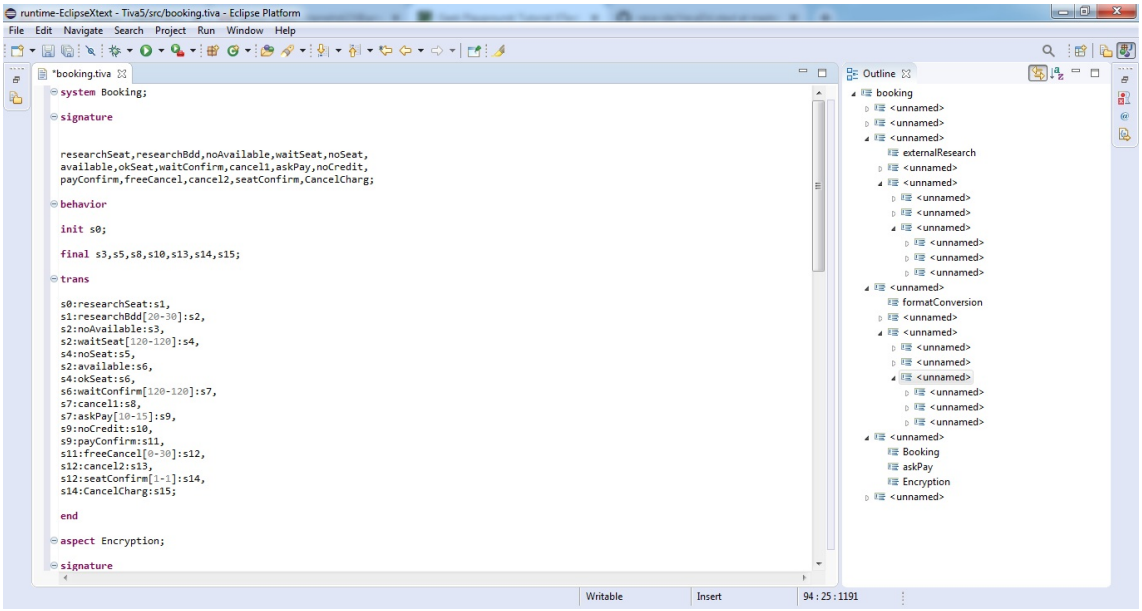

Figure 10.1: TiVA DSL

10.1.2 TiVA Core

TiVA Core (java application) transforms the edited (and saved) DSL model with TiVA DSL into a network of timed automata (xta file) as described in Chapter 8.

As is shown in the figure (Figure 10.2), once this transformation is done, the designer is not obliged to use an external tool to make verifications, since the uppaal engine is encapsulated with our TiVA Core. The designer will get the verification results in the TiVA Core, nevertheless, one has the possibility to download the xta file (see B for the full xta format of our case study), and perform graphical simulations using the graphical interface of UPPAAL tool.

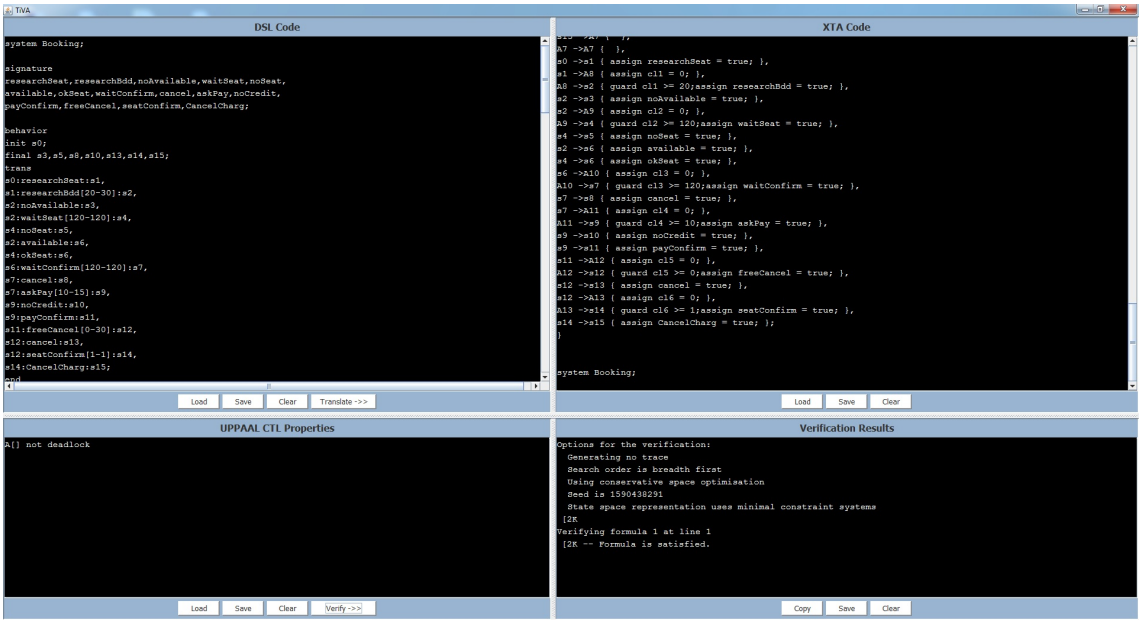

Figure 10.2: TiVA Core

10.2 Evaluation

The scalability of our modeling is evaluated in terms of the number of aspects in the system, given that for any modeling we have only one basic system. The evaluation of our case study was made on aspects having 10 states (in the DSL), woven on a basic system of 15 states (in the DSL). Table 10.1 presents the verification times (in seconds) for the *Deadlock* property, and Table 10.2 presents the results relating to the *The delay of execution* property. These results were obtained on a laptop with a Core 2 Duo processor (2.20 G. Hertz), and 4.0 G. bytes of RAM. Note that a time which is equal to zero represents a negligible time. We note that for the two evaluated properties, for a system of 2000 woven aspects, the times remain acceptable and the verification is supported.

In [82], a folder of examples is provided with the TiVA Framework; examples of simple weaving, multiple weaving, weaving with multiple occurrences of a join point, weaving with the use of wildcards, etc. Further, an example of a model of a system containing a loop is also provided, this example models a very important notion, which is the notion of timeout. Figure 10 shows that the system in state $s0$ waits until 5 uot, so that the user identifies himself, if the latter does not enter his password after 5 uot the system returns to the initial state $s0$.

Example 10 (Model of a system with a timeout)**.**

```
1 ...
2 s0:enterPass[0-5]:s00,
3 s0:reInit[5-5]:s0,
4 ...
```

Table 10.1: Evaluation of *Deadlock* property

Number of aspects	Verification
1	00.00 sec
10	00.00 sec
100	00.219 sec
200	00.717 sec
300	01.498 sec
400	02.574 sec
500	04.041 sec
800	10.218 sec
1000	16.63 sec
2000	88.827 sec

Table 10.2: Evaluation of *Delay of execution* property

Number of aspects	Verification
1	00.00 sec
10	00.00 sec
100	00.125 sec
200	00.422 sec
300	00.905 sec
400	01.544 sec
500	02.434 sec
800	06.116 sec
1000	10.171 sec
2000	67.049 sec

Chapter 11

Comparison with the state of the art

The tables 11.1 and 11.2 compare our approach according to the criteria presented in the state of the art chapter.

On the other hand, in the state of the art, many works apply formal models to channel the use of the Aspect-Oriented Approach. In [32] authors studied aspect categories and how to reason and ensure properties on aspect-oriented programs, for this, they used a language independent semantics framework to formally define several aspects categories: observers, aborters, confiners and weak intruders. Similarly, Katz in [49] gives the categories of aspects: spectative aspects, regulative aspects and weakly invasive aspects. For each category, Katz indicates which standard classes of properties are preserved. Goldman in [39] presents an automatic and modular way to answer the two questions. The first question is whether each of the aspects by itself is correct when woven alone into a suitable base system. The second question is whether the guarantee of some aspect can be invalidated as a result of weaving it together with additional aspects into the same base system. In [87] authors present an evaluation study on applying aspect-oriented modeling concepts in UPPAAL timed automata. The study is focusing on the modeling and verification effort that can be reduced when applying explicit aspect-oriented structuring principles in model construction. In [42] authors present an approach using UPPAAL to detect interferences problems between aspects, however, the authors do not give a tool automating the transformation process.
The majority of the cited works study the problems of interferences of aspects. As to the best of our knowledge, our approach is the first exploiting formal methods for software systems with time constraints, to investigate both the preservation of delays after the weaving of aspects, and the detection of aspects weaving into inappropriate places in a basic system.

Table 11.1: Comparison with the state of the art - Part 1

Approach	Our approcach (TiVA)
Input langage	TiVA DSL
Formalism	Timed automata
Verification tool	UPPAAL
Tool automating the approach	TiVA Framework
Verified issue	Behavior/AOP Evolution

Table 11.2: Comparison with the state of the art - Part 2

Approach	Our approach (TiVA)
Verified properties	All CTL properties
Design/Runtime	Design
Component model	Independent
Domain	General
Hierarchical or flat	Hierarchical
Symmetry component/aspect	Yes
Inter-component/Intra-component	Inter-component & Intra-component

Chapter 12

Conclusion and perspectives

The objective of this thesis is to verify one of the problems that can arise during the synergy of component-based development and aspect-oriented development. Indeed, we have presented a formal approach for the formal modeling and checking of timed aspect-oriented evolution for component-based development. A system is first described in a Domain Specific Language. It is then translated into a network of timed automata. Finally, this network of timed automata is the input language of the UPPAAL model checker to simulate and verify the behavior of the system.We have developed a set of tools, TiVA Framework, that supports all these phases.

As to the best of our knowledge, our approach is the first exploiting formal methods for component-based systems with time constraints, to investigate both the preservation of delays after the weaving of aspects, and the detection of aspects weaving into inappropriate places in a basic system.

A new version of TiVA Framework, on which we are currently working, offers the possibility of starting the specification directly from the basic components of the system, allowing thereafter, to generate the model of the final system automatically by a product operation of automata. Further, this version also allows to support the asynchronous communication mode between the components of the system.

Moreover, the new version of TiVA DSL includes the notion of Timeout as a predefined concept in the DSL, this is of great interest for the specification and verification of web services in particular, which represent a typical application domain for component-based systems.

One of the future perspectives is to check aspect-oriented evolution at runtime. On the other hand, we plan to provide the code generation directly from our specification.

Bibliography

[1] TiVA project. `https://hariati.github.io/tiva-framework/`.

[2] Jonathan Aldrich. Open modules: Modular reasoning about advice. *ECOOP 2005-Object-Oriented Programming*, pages 734–734, 2005.

[3] Jonathan Aldrich. A formal model of modularity in aspect-oriented programming. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development*, pages 21–25, 2011.

[4] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[5] Robert J Allen. A formal approach to software architecture. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 1997.

[6] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[7] Farhad Arbab and Farhad Mavaddat. Coordination through channel composition. In *Coordination Models and Languages*, pages 275–297. Springer, 2002.

[8] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking component composability. In *Software Composition*, volume 4089, pages 18–33. Springer, 2006.

[9] Nikhil Barthwal and Murray Woodside. Efficient evaluation of alternatives for assembly of services. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[10] Remi Bastide and Eric Barboni. Software components: A formal semantics based on coloured petri nets. *Electronic Notes in Theoretical Computer Science*, 160:57–73, 2006.

[11] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. in : Formal methods for the design of real-time systems. In *Springer Berlin Heidelberg*, 2004.

[12] Antoine Beugnard, J.-M. Jezequel, and et al. Plouzeau, Noï¿½l. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[13] Grady Booch, James Rumbaugh, Ivar Jacobson, et al. *Le guide de l'utilisateur UML*, volume 3. Eyrolles, 2000.

[14] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[15] Johan Brichau, Maurice Glandrup, Siobhan Clarke, and Lodewijk Bergmans. Advanced separation of concerns. In *European Conference on Object-Oriented Programming*, pages 107–130. Springer, 2001.

[16] 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*, 36(11-12):1257–1284, 2006.

[17] E. Cambronero, J. C. Okika, and A. P. Ravn. Analyzing web service contracts. In *Mobile Ubiquitous Computing, Systems, Services and Technologies. UBICOMM'07*, 2007.

[18] M. E. Cambronero, G. Dï¿½az, V. Valero, and E. Martï¿½nez. Validation and verification of web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49, 2011.

[19] M Emilia Cambronero, Juan J Pardo, Gregorio Diaz, and Valentin Valero. Timed automata for web services verification. In *Proceedings of the 6th WSEAS international conference on Applied computer science*, pages 531–536. World Scientific and Engineering Academy and Society (WSEAS), 2006.

[20] Z. Chen. Formal analysis and verification of web service compositions with timing constraints. *International Journal of Advancements in Computing Technology*, 4(17), 2012.

[21] Pedro J Clemente, Juan Hernández, José Luis Herrero, Juan Manuel Murillo, Fernando Sánchez, et al. Aspect-orientation in the software lifecycle: Fact and fiction. *Filman et al.[426]*, pages 407–423, 2005.

[22] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619, pages 331–346. Springer, 2003.

[23] OMG CORBA and IIOP Specification. Object management group. *Joint revised submission OMG document orbos/99-02*, 1999.

[24] Thomas Cottenier and Tzilla Elrad. Validation of context-dependent aspect-oriented adaptations to components. *WCOP, Oslo*, 2004.

[25] Ivica Crnkovic, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. A classification framework for component models. *Software Engineering Research and Practice in Sweden*, page 3, 2007.

[26] Luca De Alfaro and Thomas A Henzinger. Interface-based design. *Engineering Theories of Software-intensive Systems*, 195:83–104, 2005.

[27] R. De Nicola, G. Ferrari, and G. Meredith. Eds. In *The 6th International Conference in Coordination Models and Languages*. Springer-Verlag, 2004.

[28] LG DeMichiel, L Ümit Yalçinalp, and S Krishnan. Sun microsystems-enterprise javabeanstm specification, 2000.

[29] Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard. Automating the building of software component architectures. *Software Architecture*, pages 228–235, 2006.

[30] G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Verification of web services with timed automata. *Electronic Notes in Theoretical Computer Science*, 157(2):19–34, 2006.

[31] Gregorio Diaz, M Emilia Cambronero, Juan J Pardo, Valentin Valero, and Fernando Cuartero. Model checking techniques applied to the design of web services. *CLEI Electron. J*, 10(2), 2007.

[32] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. *Science of Computer Programming*, 77(3):393–422, 2012.

[33] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of computation orchestration via timed automata. In *Formal Methods and Software Engineering. Springer Berlin Heidelberg*, 2006.

[34] Scott Duncan. Component software: Beyond object-oriented programming. *Software Quality Professional*, 5(4):42, 2003.

[35] Ondrej Galik and Tomas Bures. Generating connectors for heterogeneous deployment. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 54–61. ACM, 2005.

[36] YAN Gaogao, ZHU Xue-Yang, and YAN Rongjie et al. Formal throughput and response time analysis of marte models. In *International Conference on Formal Engineering Methods*, 2014.

[37] Ning Ge and Marc Pantel. Time properties verification framework for uml-marte safety critical real-time systems. In *European Conference on Modelling Foundations and Applications*, 2012.

[38] Lars Gesellensetter and Sabine Glesner. Only the best can make it: Optimal component selection. *Electronic Notes in Theoretical Computer Science*, 176(2):105–124, 2007.

[39] Max Goldman, Emilia Katz, and Shmuel Katz. Maven: modular aspect verification and interference analysis. *Formal Methods in System Design*, 37(1):61–92, 2010.

[40] N. Guermouche and C. Godart. Timed model checking based approach for web services analysis. In *Web Services, 2009. ICWS 2009. IEEE International Conference*, 2009.

[41] N. Guermouche and C. Godart. Analyse de la compatibilite des services web temporises. In *Actes du XXVIIIï¿½ congrï¿½s INFORSID, Marseille.*, 2010.

[42] Abdelhakim Hannousse, Rémi Douence, and Gilles Ardourel. Static analysis of aspect interaction and composition in component models. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, pages 43–52, 2011.

[43] Mehdi Hariati. Formal verification issues for component-based development. *Informatica*, 44(4), 2020.

[44] Mehdi Hariati and Djamel Meslati. Review on formal approaches for the verification of aspect-oriented adaptations of component-based systems. *International Journal of Software Engineering and Its Applications*, pages 39–44, 2017.

[45] George T Heineman and William T Councill. Component-based software engineering. *Putting the pieces together, addison-westley*, page 5, 2001.

[46] C. Heinzemann and S. Becker. Executing reconfigurations in hierarchical component architectures. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering. ACM*, 2013.

[47] N. Ibrahim, V. Alagar, and M. Mohammad. Specification and verification of context-dependent services. In *arXiv preprint arXiv:1108.2349*, 2011.

[48] SURYADEVARA Jagadish, SECELEANU Cristina, and MALLET Frederic et al. Verifying marte/ccsl mode behaviors using uppaal. In *International Conference on Software Engineering and Formal Methods*, 2013.

[49] Shmuel Katz. Aspect categories and classes of temporal properties. In *Transactions on aspect-oriented software development I*, pages 106–134. Springer, 2006.

[50] R. Kazhamiakin, P. Pandya, and M. Pistore. Timed modelling and analysis in web service compositions. In *Availability, Reliability and Security. ARES 2006. The First International Conference*, 2006.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. *ECOOP 2001ï¿½Object-Oriented Programming*, pages 327–354, 2001.

[52] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[53] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97ï¿½Object-oriented programming*, pages 220–242, 1997.

[54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[55] Ramnivas Laddad. *AspectJ in action: practical aspect-oriented programming.* Dreamtech Press, 2003.

[56] M. Lallali, F. Zaidi, and A. Cavalli. Timed modeling of web services composition for automatic testing. In *Signal-Image Technologies and Internet-Based System. SITIS'07. Third International IEEE Conference*, 2007.

[57] M. Lallali, F. Zaidi, A. Cavalli, and I. Hwang. Automatic timed test case generation for web services composition. In *On Web Services. ECOWS'08. IEEE Sixth European Conference*, 2008.

[58] K. G. et al. Larsen. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[59] Kim G Larsen and Arne Skou. Bisimulation through probabilistic testing (preliminary report). In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 344–352. ACM, 1989.

[60] Nancy A Lynch and Mark R Tuttle. An introduction to input/output automata. 1988.

[61] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. *Software Engineeringï¿½ï¿½ESEC'95*, pages 137–153, 1995.

[62] Leonardo Mariani and Mauro Pezze. A technique for verifying component-based software. *Electronic Notes in Theoretical Computer Science*, 116:17–30, 2005.

[63] J. Mei, H. Miao, Q. Xu, and P. Liu. Modeling and verifying web service applications with time constraints. In *Computer and Information Science (ICIS) IEEE/ACIS 9th International Conference*, 2010.

[64] Philipp Meier, Samuel Kounev, and Heiko Koziolek. Automated transformation of component-based software architecture models to queueing petri nets. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 339–348. IEEE, 2011.

[65] Bertrand Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003.

[66] Bertrand Meyer, Christine Mingins, and Heinz Schmidt. Providing trusted components to the industry. *Computer*, 31(5):104–105, 1998.

[67] B. Morin, O. Barais, G. Nain, and J. M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society*, 2009.

[68] Juan Murillo, Juan Hernández, Fernando Sánchez, and Luis Álvarez. Coordinated roles: Promoting re-usability of coordinated active objects using event notification protocols. *Coordinatio Languages and Models*, pages 647–647, 1999.

[69] MENAD Nadia, DHAUSSY Philippe, and DREY Zoe et al. Towards a transformation approach of timed uml marte specifications for observer-based formal verification. *Computing and Informatics*, 35(2), 2016.

[70] Dong Ha Nguyen and Mario Südholt. Property-preserving evolution of components using vpa-based aspects. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 613–629. Springer, 2007.

[71] Dirk Niebuhr and Andreas Rausch. A concept for dynamic wiring of components: correctness in dynamic adaptive systems. In *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 101–102. ACM, 2007.

[72] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *2006 30th Annual IEEE/NASA Software Engineering Workshop*, pages 133–141. IEEE, 2006.

[73] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA*, pages 133–141. IEEE, 2006.

[74] Jennifer Pérez, Nour Ali, Jose Carsí, and Isidro Ramos. Designing software architectures with an aspect-oriented architecture description language. *Component-Based Software Engineering*, pages 123–138, 2006.

[75] P. Pettersson, T. Seceleanu, and S. E Ellevseth. Wind turbine system: An industrial case study in formal modeling and verification. In *Formal Techniques for Safety-Critical Systems: Second International Workshop*, 2013.

[76] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE transactions on Software Engineering*, 28(11):1056–1076, 2002.

[77] G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the semantics and verification of bpel4ws. *Electronic Notes in Theoretical Computer Science*, 151(2):33–52, 2006.

[78] Irum Rauf, Faezeh Siavashi, Dragos Truscan, and al. An integrated approach for designing and validating rest web service compositions. In *WEBIST (1)*, 2014.

[79] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and F Plasil. The common component modeling example. *Lecture notes in computer science*, 5153, 2008.

[80] Andrï¿½ Restivo and Ademar Aguiarl. Aspects: Conflicts and interferences a survey. In *2iï¿½ Conferiï¿½ncia em Metodologias de Investigaiï¿½iï¿½o Cientiï¿½fica*, 2007.

[81] M. Rouached and C. Godart. Requirements-driven verification of wsbpel processes. In *In Web Services. ICWS 2007. IEEE International Conference*, 2007.

[82] Q. Z. Sheng, Z. Maamar, L. Yao, C. Szabo, and S. Bourne. Behavior modeling and automated verification of web services. *Information Sciences*, 258:416–433, 2014.

[83] Marianne Simonot and Maria-Virginia Aponte. Une approche formelle de la reconfiguration dynamique. *L'Objet*, 14(4):73–102, 2008.

[84] Guido Söldner and Rüdiger Kapitza. Aoci: an aspect-oriented component infrastructure. In *Proc. of the 12th Int. Workshop on Component Oriented Programming (WCOP 2007)*, pages 53–58, 2007.

[85] Kun Tian, Kendra Cooper, Kang Zhang, , and Huiqun Yu. A classification of aspect composition problems. In *Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2009.

[86] Kun Tian, Kendra Cooper, Kang Zhang, and Huiqun Yu. A classification of aspect composition problems. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 101–109. IEEE, 2009.

[87] Jüri Vain, Dragos Truscan, Junaid Iqbal, and Leonidas Tsiopoulos. On the benefits of using aspect-orientation in uppaal timed automata. In *2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions)(ICTUS)*, pages 84–91. IEEE, 2017.

[88] Pakorn Waewsawangwong. A constraint architectural description approach to self-organising component-based software systems. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 81–83. IEEE, 2004.

[89] Sara Williams and Charlie Kindel. The component object model: A technical overview. *Dr. Dobbs Journal*, 356:356–375, 1994.

[90] Gaoyan Xie. Decompositional verification of component-based systems-a hybrid approach. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 414–417. IEEE Computer Society, 2004.

[91] Tao Xie, Jianjun Zhao, Darko Marinov, and David Notkin. Automated test generation for aspectj programs. In *AOSD 2005 Workshop on Testing Aspect-Oriented Programs, Chicago*, pages 1–6, 2005.

[92] Jia Xu, Hridesh Rajan, and Kevin Sullivan. Understanding aspects via implicit invocation. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 332–335. IEEE Computer Society, 2004.

[93] Barbora Zimmerova. Formal analysis of component-based systems in view of comp. interactions. In *Proceedings of the International Research Training Groups Workshop 2006*, volume 3, page 34. GITO mbH Verlag, 2006.

[94] Barbora Zimmerová. *Modelling and formal analysis of component-based systems in view of component interaction.* PhD thesis, Masarykova univerzita, Fakulta informatiky, 2008.

[95] Barbora Zimmerova et al. Component placement in distributed environment wrt component interaction. In *Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, page 260. FIT VUT Brno, Czech Republic, 2006.

[96] Barbora Zimmerova and Pavlına Vareková. Reflecting creation and destruction of instances in cbss modelling and verification. In *Proceedings of the Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICSï¿½ï¿½ 07)*, pages 257–264, 2007.

Appendix A

The DSL of our case study

```
system Booking;

signature
researchSeat,researchBdd,noAvailable,waitSeat,noSeat,
available,okSeat,waitConfirm,cancel,askPay,noCredit,
payConfirm,freeCancel,seatConfirm,CancelCharg;

behavior
init s0;
final s3,s5,s8,s10,s13,s14,s15;
trans
s0:researchSeat:s1,
s1:researchBdd[20-30]:s2,
s2:noAvailable:s3,
s2:waitSeat[120-120]:s4,
s4:noSeat:s5,
s2:available:s6,
s4:okSeat:s6,
s6:waitConfirm[120-120]:s7,
s7:cancel:s8,
s7:askPay[10-15]:s9,
s9:noCredit:s10,
s9:payConfirm:s11,
s11:freeCancel[0-30]:s12,
s12:cancel:s13,
s12:seatConfirm[1-1]:s14,
s14:CancelCharg:s15;
end

aspect Encryption;

signature
inputData, encryptData, outputRes;

behavior
init s0;
final s3;
trans
s0:inputData:s1:trigger,
s1:encryptData[13-18]:s2,
s2:outputRes:s3:stop;
end

aspect encryptionLan;

signature
inputData, encryptDataLan, outputRes;

behavior
init s0;
final s3;
```

```
trans
s0:inputData:s1:trigger,
s1:encryptDataLan[12-14]:s2,
s2:outputRes:s3:stop;
end

aspect encryptionWifi;

signature
inputData, encryptDataWifi, outputRes;

behavior
init s0;
final s3;
trans
s0:inputData:s1:trigger,
s1:encryptDataWifi[11-19]:s2,
s2:outputRes:s3:stop;
end

aspect externalResearch;

signature
inputData, searchData, outputRes;

behavior
init s0;
final s3;
trans
s0:inputData:s1:trigger,
s1:searchData[10-15]:s2,
s2:outputRes:s3:stop;
end

aspect formatConversion;

signature
inputData, convertData, outputRes;

behavior
init s0;
final s3;
trans
s0:inputData:s1:trigger,
s1:convertData[5-10]:s2,
s2:outputRes:s3:stop;
end

aspect cancelReport;

signature
inputData, makeReport, outputRes;

behavior
init s0;
final s3;
trans
s0:inputData:s1:trigger,
s1:makeReport[2-7]:s2,
s2:outputRes:s3:stop;
end

// Examples of weavings, but some must be checked separately

// Simple weaving of a single aspect on a single operation
Weaving (Booking:researchBdd[20-30]:externalResearch:after),

// Simple weaving of a single aspect on a single operation: case several occurrences of
   this operation
Weaving (Booking:cancel:cancelReport:after),
```

```
121
122 // Simple weaving of a single aspect on several operations: use of wildcards in the
      weaving expression
123 Weaving (Booking:*Pay*:Encryption:before);
124
125 // Multiple weavings of several aspects on the same operation using the Mutex adpater
126 Adapter(Booking:askPay[10-15]:encryptionLan:encryptionWifi:before:mutex),
127
128 // Multiple weavings of several aspects on the same operation using the Prec adpater
129 Adapter (Booking:researchBdd[20-30]:externalResearch:formatConversion:after:prec);
```

Appendix B

The XTA code of our case study (generated by the TiVA Core): case of a simple weaving

```
chan
Ch1;
clock g;

process Booking(  ) {

clock
cl1,
cl2,
cl3,
cl4,
cl5,
cl6;

bool
researchSeat=false,
researchBdd=false,
noAvailable=false,
waitSeat=false,
noSeat=false,
available=false,
okSeat=false,
waitConfirm=false,
cancel=false,
askPay=false,
noCredit=false,
payConfirm=false,
freeCancel=false,
seatConfirm=false,
CancelCharg=false;

state
s0  ,
s3  ,
A1  ,
s5  ,
A2  ,
s8  ,
A3  ,
s10  ,
A4  ,
s13  ,
A5  ,
s14  ,
A6  ,
```

```
s15 ,
A7 ,
s1 ,
s2 ,
A8 {cl1 <=30},
s4 ,
A9 {cl2 <=120},
s6 ,
s7 ,
A10 {cl3 <=120},
s9 ,
A11 {cl4 <=15},
s11 ,
s12 ,
A12 {cl5 <=30},
A13 {cl6 <=1},
A16 ;

urgent
s0,
s3,
s5,
s8,
s10,
s13,
s14,
s15,
s1,
s2,
s4,
s6,
s7,
s9,
s11,
s12;

init s0;

trans
s3->A1 { },
A1 ->A1 { },
s5 ->A2 { },
A2 ->A2 { },
s8 ->A3 { },
A3 ->A3 { },
s10 ->A4 { },
A4 ->A4 { },
s13 ->A5 { },
A5 ->A5 { },
s14 ->A6 { },
A6 ->A6 { },
s15 ->A7 { },
A7 ->A7 { },
s0 ->s1 { assign researchSeat = true; },
s1 ->A8 { assign cl1 = 0; },
s2 ->s3 { assign noAvailable = true; },
s2 ->A9 { assign cl2 = 0; },
A9 ->s4 { guard cl2 >= 120;assign waitSeat = true; },
s4 ->s5 { assign noSeat = true; },
s2 ->s6 { assign available = true; },
s4 ->s6 { assign okSeat = true; },
s6 ->A10 { assign cl3 = 0; },
A10 ->s7 { guard cl3 >= 120;assign waitConfirm = true; },
s7 ->s8 { assign cancel = true; },
s7 ->A11 { assign cl4 = 0; },
A11 ->s9 { guard cl4 >= 10;assign askPay = true; },
s9 ->s10 { assign noCredit = true; },
s9 ->s11 { assign payConfirm = true; },
s11 ->A12 { assign cl5 = 0; },
A12 ->s12 { guard cl5 >= 0;assign freeCancel = true; },
```

```
s12 ->s13 { assign cancel = true; },
s12 ->A13 { assign cl6 = 0; },
A13 ->s14 { guard cl6 >= 1;assign seatConfirm = true; },
s14 ->s15 { assign CancelCharg = true; },
A8 ->A16 { guard cl1 >= 20;sync Ch1!; },
A16 ->s2 { sync Ch1?; };
}


process externalResearch(  ) {

clock
cl7;

bool
inputData=false,
searchData=false,
outputRes=false;

state
s0 ,
s3 ,
A14 ,
s1 ,
s2 ,
A15 {cl7<=15};

urgent
s3,
s1,
s2;

init s0;

trans
s3->A14 {  },
A14 ->A14 {  },
s0 ->s1 { sync Ch1?;assign inputData = true; },
s1 ->A15 { assign cl7 = 0; },
A15 ->s2 { guard cl7 >= 10;assign searchData = true; },
s2 ->s3 { sync Ch1!;assign outputRes = true; };
}


system Booking, externalResearch;
```

)