UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATHÉMATIQUES

Année 2011

N° attribué par la bibliothèque

— — — — — — — — — — —

A VPA-based aspect language - an approach to software composition using non-regular behavioral protocols

THÈSE DE DOCTORAT
Discipline : Informatique
Spécialité: Génie logiciel

Présentée
et soutenue publiquement par

Dong-Ha NGUYEN

Le 21 Octobre 2011, devant le jury ci-dessous

Rapporteurs:	Benoît Baudry, CR INRIA	INRIA Rennes - Bretagne Atlantique
	Antoine Beugnard, Pr	Telecom Bretagne
Examinateurs:	Christian Attiogbé, Pr	Université de Nantes
	Isabelle Borne, Pr	Université de Bretagne-Sud
	Pierre Cointe, Pr	Ecole des Mines de Nantes
	Mario Südholt, Pr	Ecole des Mines de Nantes

Directeur de thèse: Pierre Cointe
Co-encadrant: Mario Südholt

ED : .....

Abstract

Currently, there are only few AOP systems that exploit protocol-based pointcut languages in order to enable declarative aspect definitions and provide support for reasoning over properties of AO programs. Furthermore, most approaches to protocol-based software evolution only support regular protocols which are limited in expressiveness. The goal of this thesis is to develop an aspect language that is not only more expressive but also amenable to property analysis and verification. Our aspect language addresses these limitations by extending the original framework with more specific language support for non-regular protocols as well as providing techniques for the analysis of properties of components. Concretely, this thesis provides four contributions. First, we have defined the VPA-based aspect language, which provides an aspect model on top of protocols defined using the class of visibly pushdown automata (VPAs). Second, we have used aspects over interaction protocols of software components in order to define and analyze fundamental correctness properties of components. Third, we have considered how to harness existing model checkers to verify systems that are modified by VPA-based aspects. Fourth, we have shown how VPA-based aspects can be useful for the definition of different functionalities in two different application domains: the management of nested login sessions, and the management of queries in P2P-based grid systems.

Key words: aspect-oriented programming, visibly pushdown automata, interaction protocols, software evolution, component-based systems, model checking, peer-to-peer systems, correctness by construction

Résumé

Actuellement, il n'existe que peu de systèmes qui exploitent des langages de coupes basés sur des protocoles afin d'offrir des définitions d'aspects déclaratifs et de permettre de raisonner sur des propriétés de programmes orientés aspect. Par ailleurs, la plupart des approches à l'évolution des logiciels basés sur des protocoles ne soutiennent que des protocoles réguliers dont l'expressivité est limitée. L'objectif de cette thèse est de développer un langage d'aspects qui est non seulement plus expressif, mais aussi favorable à l'analyse et la vérification des propriétés compositionnelles. Notre langage d'aspect adresse de ces limitations par un langage basé sur des protocoles non réguliers ; nous fournissons, en outre, des techniques pour l'analyse des propriétés des composants qui sont modifiés à l'aide du langage. Concrètement, cette thèse a donné lieu à quatre contributions principales. Nous avons d'abord défini un langage d'aspects manipulant des protocoles définis à l'aide d'automates à pile visible (APV). Ensuite, nous avons appliqués ces aspects à l'analyse de propriétés compositionnelles fondamentales d'interactions entre composants. Troisièmement, nous avons examiné la technique de la vérification des modèles afin de vérifier les systèmes modifiés par des aspects fondés sur des APV. Quatrièmement, nous avons utilisé les aspects APV pour la définition des différentes fonctionnalités dans deux domaines d'application différents : la gestion de sessions de connexion imbriquées, et la gestion de requtes dans les systèmes pair-à-pair.

Mots-clés : programmation orientée aspect, automates à pile visible, protocoles d'interaction, évolution du logiciel, systèmes à base de composants, model checking, systèmes pair-à-pair, correction par construction

Acknowledgements

The research presented in this thesis has been supported by AOSD-Europe, the European Network of Excellence in AOSD (`http://www.aosd-europe.net`)

First and foremost, I would like to deeply thank my supervisor Mario Südholt for his great patience, his guidance and support throughout the production of this research and thesis. I appreciate all his contributions of time and ideas to this research, to our co-written papers and to my thesis. Without his continous support and pushing, I could not have finished my thesis successfully.

I would also like to thank my thesis director, Pierre Cointe, for helping me with several administrative procedures and for serving on my thesis committee. I also thank my thesis reviewers: Benoît Baudry and Antoine Beugnard for their time, interest, and helpful remarks. I would also like to thank Christian Attiogbé and Isabelle Borne for serving on my thesis committee.

I owe my sincere gratitude to Thao Dang for giving me encouragement and advice during the final stage of this PhD. My discussions with her gave me an insight into the evaluation process and helped me greatly to gain confidence and determination to press forward.

I would like to thank my grandma, dad, mom, and sister for all their love, unwavering support and encouragement throughout the years. If it wasn't for them, I wouldn't be able to finish this thesis. I also thank my extended family members who always care about me and take good care of my mom when I am so far away.

Last, but not least, I would like to express my heart-felt gratitude to my loving husband, Hoang, for supporting me and enduring all the stress I put on him through all the years I worked on my thesis. Hoang had spent a lot of his time and energy to take care of our baby daughter Misha so that I could find some time to finish my thesis. I am so blessed to have him by my side in this long dissertation journey. And thanks to Misha for bringing me lots of joys and never failing to lift my spirits. I love you so much.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Aspect-Oriented Software Development (AOSD) aims at the systematic treatment of functionalities, so-called crosscutting concerns, that cannot reasonably be modularized using traditional program and application structuring means, such as object-oriented programming and component-based software development. As crosscutting concerns — such as execution of queries in P2P systems, security and transactional concurrency control in business information systems — abound in large-scale software systems, a large number of AO mechanisms have been proposed by now for the modularization of crosscutting concerns, especially for component-based systems. However, almost all of these approaches provide programmatic means without any enforceable guarantees on the correctness of aspect application. These aspect languages and systems are frequently used for the design and implementation of evolution scenarios for legacy software systems: the lack of correctness guarantees for aspects is frequently seen as a major impediment to their generalized use.

One popular means to introduce means for the formal analysis and verification of correctness properties of software systems, is the use of behavioral protocols that make explicit parts of the semantics of software systems and can be used for property analysis and enforcement. In contrast to the large majority of language-based approaches to software evolution that are based on Turing-complete mechanism, protocol-based approaches typically rely on more restricted formalisms that support the more declarative definition of constraints on the execution of software systems and enable semi-automatic or automatic analysis, validation and verification mechanisms to be used.

While protocols have already been widely used to support declarative programmatic access to components as well as automatic reasoning about component properties, there are only few AOP systems that exploit protocol-based pointcut languages in order to enable declarative aspect definitions and provide support for reasoning over properties of AO programs. Furthermore, most approaches to protocol-based software evolution, independent from whether they use aspects or not, only support regular, that is, finite-state based protocols. These protocols have the advantage that many of their properties can be efficiently and automatically analyzed and verified, especially using model checking techniques. Harnessing the same advantages, a fair number of aspect systems with regular pointcut languages have been proposed, most notably regular aspects by Douence et al. [47, 48] and tracematches by Allan et al. [17].

Regular protocols are, however, limited in expressiveness: they cannot express, in particular, computations that require unbounded counting of events. Furthermore, they do not support arbitrary deep well-balanced nested expressions that occur frequently in many ap-

plication domains, for example, in form of matching calls and returns or query and replies. Such execution structures are supported by more expressive types of protocols, in particular context-free protocols, but very few component-based and aspect-oriented systems have been proposed that feature non-regular protocols, the main reason being that the analysis and verification techniques for non-regular protocols are few and much less efficient than those for regular protocols.

In this thesis we have explored the definition of aspect-oriented programming techniques on top of non-regular behavioral protocols, more concretely a recent variant of non-regular protocols, defined using the so-called visibly pushdown automata (introduced in 2004 by Alur and Madhusudan [21]). We have shown how to integrate them on the language-level, provided new means for the analysis and verification of aspect properties, as well as how to develop approximations in order to harness them with existing model checking tools.

1.1 Context

Existing AOP approaches that feature pointcuts that are capable of matching protocols include regular aspects by Douence et al. [47, 48], tracematches by Allan et al. [17], stateful aspects in JAsCo by Vanderperren et al. [125], and tracecuts by Walker and Viggers [128]. The first three approaches only support regular protocols. The fourth approach supports protocols described by context-free grammar, especially those that involve properly nested structures. However, this feature impedes analysis support because basic decision problems such as inclusion, emptiness or universality are undecidable for context-free languages [67].

The aspect language introduced in this thesis have been developed based on the generic framework for AOP [47]. This framework provides a foundation for the development of AO languages that support history-based pointcuts and the analysis for interaction between aspects. The syntax and semantics defined by this framework have been reused in the definition of our aspect language. However, this framework only supports regular aspects and does not consider analysis techniques for compositional properties in component-based systems. Our aspect language addresses these limitations by extending the original framework with more specific language supports for non-regular protocols as well as providing techniques for the analysis of properties of components.

There have been quite a number of AO approaches, *e.g.,* JBoss/AOP [69], SpringAOP [12], JAsCo [120], CaesarJ [91, 23], that aim at component-based applications. However, among these approaches, only JAsCo provides explicit support for defining aspects over protocols (which are finite-state based protocols). We aim for an aspect language that provides support for aspects over non-regular protocols of components. Note that, we consider components with explicit protocols, *i.e.,* components of which protocols are exposed through the interface, to be applications of our aspect language. This restriction permits non-invasive aspect modifications to components. Furthermore, this allows us to attempt to analyze at the protocol level the properties of the component-based system subject to aspect-based evolution.

An aspect language that supports non-regular pointcuts is very useful for applications that require more expressive than regular protocols. Peer-to-peer (P2P) systems are one particular application domain that involves non-regular protocols defined on nested well-balanced contexts such as the recursive query protocol. We have explored further P2P-based applications and frameworks including JXTA[8], a Java technology that supports the creation of P2P applications, and three P2P-based grid systems, Jalapeno[122], Juxta-CAT[108], and

OurGrid[28], as they also use a number of protocols that are potential targets of aspects defined by our language.

1.2 Contributions

The goal of this thesis is to develop an aspect language that is not only more expressive (than regular ones) but also amenable to property analysis and verification, especially for component-based applications that are subject to evolution by aspects. For this purpose, we have developed an advanced history-based aspect language that features a protocol-based pointcut language and thus enables declarative aspect definitions. Furthermore, we have developed tool support for the execution and analysis of the corresponding aspect systems. Finally, we have harnessed our aspect language in different application domains. Concretely, this thesis has resulted in the following four contributions.

First, based on an extensive survey of related work in AOSD and CBSE, we have defined a new aspect language, called the VPA-based aspect language, that provides an aspect model on top of protocols defined using the class of visibly pushdown automata (VPAs). This language enables the declarative definition of evolution scenarios that require the manipulation of nested well-balanced execution structures. The language features VPA-based pointcuts and provides, in particular, constructors for the declarative definition of pointcuts based on regular and non-regular structures. Furthermore, our language enables aspect properties, notably interaction properties to be analyzed formally. We present a semantics of our language by extending an existing framework of regular aspect languages [47]. We have also extended and developed the technique for detecting automatically potential interactions among VPA-based aspects. Finally, we have realized a library called VPAlib that provides an, as to our knowledge first, implementation of essential data structures and operations for the VPA. This library is essential to enable the construction and analysis of VPA-based aspects. We have applied the library to some analysis problems involving VPA-based aspects.

Second, we have used aspects over interaction protocols of software components in order to define and analyze fundamental correctness properties of components, including compatibility and substitutability relations that are subject to evolution using VPA-based aspects. Instead of analyzing the software system after aspect weaving, our approach defines classes of aspects that satisfy certain properties by construction.

Third, we have considered how to harness existing model checkers to verify systems that are modified by VPA-based aspects. Concretely, we have developed a model checking framework tailored for VPA-based systems. In this framework, we create an abstract model of a system using VPA-based aspects and run the model checker on this abstract model.

Fourth, we have considered the application of VPA-based aspects to several problems from two different application domains: (i) the management of nested login sessions in order to improve the security of distributed systems, and (ii) the management of queries in P2P-based grid systems. We have shown how VPA-based aspects can be useful for the definition of different functionalities in such applications.

Finally, as part of these contributions, we have also investigated the limitations of VPAs in the context of AOP and component-based programming.

A part of the contributions presented in this thesis have been published in three following peer-reviewed articles:

- D. H. Nguyen, M. Südholt, “VPA-based aspects: better support for AOP over proto-

cols", 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), Sep. 2006.

- D. H. Nguyen, M. Südholt, "Towards correct evolution of components using VPA-based aspects", Proc. of the 4th International Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07) at ECOOP, July 2007.

- D. H. Nguyen, M. Südholt, "Property-preserving evolution of components using VPA-based aspects", 9th International Conference on Distributed Objects, Middleware, and Applications (DOA'07), Nov. 2007.

The SEFM'06 paper presents the major features of our VPA-based aspect language, the VPAlib library, and the method for detecting automatically potential interactions among VPA-based aspects. The content of this paper is presented in Chapter 4 of this thesis. The RAMSE'07 paper and DOA'07 paper introduce several extensions to the VPA-based language that facilitate the definition of aspects over components and discuss different proof techniques for the preservation of compositional properties of component-based systems that are subject to protocol-modifying aspects. The contributions in these two papers are mainly presented in Chapter 6 of this thesis. The applications of VPA-based aspects and the verification of systems modified by VPA-based aspects using a model checker that are presented in the remaining two contribution chapters directly follow from the results presented in the above articles.

The analysis technique for the detection of interference among aspects presented in Chapter 4 has also been published in the following technical report of the AOSD-Europe network of excellence:

- Emilia Katz, Shmuel Katz, Wilke Havinga, Tom Staijen, Nathan Weston, Francois Taiani, Awais Rashid, Dong Ha Nguyen, Mario Südholt,"Detecting interference among aspects", Technical report AOSD-Europe Deliverable D116, AOSD-Europe Network of Excellence, Feb. 2007.

The VPAlib library has been integrated into CAPE(the Common Aspect Proof Environment) [3], which is an extensible environment for different aspect verification and analysis tools and over various aspect languages. CAPE has been designed and implemented by the group at Technion, with tools (*e.g.,* VPAlib) contributed from several AOSD-Europe sites.

I have also contributed to two technical reports which are results from various administrative activities within the AOSD-Europe network of excellence:

- Mario Südholt, Dong Ha Nguyen, Awais Rashid, "Six-monthly evaluation report of mobility activities and replanning", Technical report AOSD-Europe Milestone D79, AOSD-Europe Network of Excellence, Feb. 2007.

- Dong Ha Nguyen, Mario Südholt, "Improvement plan for mobility infrastructure", Technical report AOSD-Europe Milestone M16, AOSD-Europe Network of Excellence, July. 2007.

1.3 Outline of the Dissertation

This dissertation is divided into two parts, the first presenting related work and the second the contributions of the thesis.

Chapter 2 presents an overview of important concepts and approaches in CBSE that are strongly related to our work. We are especially interested in the use of interaction protocols as specifications that express correct communications between components. Such specifications are useful in order to reason about the properties of components and their composition. This chapter also presents approaches for the analysis and verification of component-based applications and studies on component evolution.

Chapter 3 presents three fundamental issues to AOSD that are underlying the motivation and contribution of this thesis: language support for AOP, formal methods for the analysis and verification of AO programs, and AO approaches for component-based software. We conclude the chapter with lessons that we have learned from previous work and a list of features that our aspect language should offer.

The second part presents our contributions in five chapters.

Chapter 4 presents the VPA-based Aspect Language. Here we motivates the use of visibly pushdown automata to define history-based aspect languages. We present the formal syntax and a suitable semantic framework. This chapter also introduces the library that we have implemented in order to provide analysis support for VPA-based aspects. Finally, we show how interactions among VPA-based aspects can be tackled with the support of our VPA library.

Chapter 5 presents applications of VPA-based aspects in two contexts. First, VPA-based aspects are employed to supervise access in nested login sessions in distributed systems. Second, VPA-based aspects are applied to an abstract grid system deployed on peer-to-peer network.

Chapter 6 presents our approach to prove the preservation of properties of component-based systems that are modified by VPA-based aspect. We introduce the basic model of our approach and a set of theorems that allows us to prove the preservation of properties for certain classes of aspects and component-based systems.

Chapter 7 presents our work on the application of model checking in VPA-based aspect programs. We discuss different potential model checking approaches and present a framework for model checking VPA-based aspect programs. Finally, we show some experiments we have performed with a sample VPA-based system with a model checker.

The conclusion chapter recapitulates the contributions of the dissertation and proposes directions for future work.

Part I

State of the Art

Chapter 2

Component-Based Software Engineering

2.1 Introduction

Component-based software engineering (CBSE) focuses on building large software systems from reusable components [61] to make software easier to develop and maintain. In recent years, this development paradigm has emerged as a key technology that promises to ultimately enhance software quality through improving software reusability, maintainability, extensibility, and adaptibility while reducing development cost.

Nowadays, the component-based approach has become the de-facto standard for the development of large-scale software systems. The rise of innovative Internet technologies leads to the use of a multitude of distributed component-based software and services. Those applications are often built from a variety of independent units developed by different parties. Besides, expectations on the quality of computer software become higher and thus make software even more complex than before. Many organisations adopt software outsourcing into their strategic development policies to meet the high demands of the industry at inexpensive cost. Software projects can be divided into smaller portions to be sent over to other companies to have them done and later collected to build the final products. Such development strategy is only feasible with today's component-based technologies.

Along with the emergence of CBSE in the software industry, a large number of research activities on numerous aspects of the CBSE paradigm have been conducted. This thesis especially relates to studies on three major subjects on component-based software engineering: (i) specification of components and their composition, (ii) verification and analysis of behavioral properties, and (iii) evolution of component composition.

Component specification is a subject that has got a lot of attention over the years. Several component models have been developed. Most of them propose their own standards for the specification of components, especially component interfaces. Understanding components correctly is very important when building a component-based application. Knowing what they can offer and how they should be used help developers to choose the right components and use them properly. Since components are supposed to be used by third parties, their specifications should provide sufficient details so that they can be used correctly either alone or in composition with other components. Additionally, the specification of the composition of components also provides valuable information for learning the behavior of components

through behavioral protocols. An important part of our work has been based on components featuring explicit protocols in their interfaces.

Verification and analysis techniques would help to ensure correctness and reliability of component-based software. Although the component technologies have emerged significantly over recent years, verification and analysis techniques for component-based software are still scarce. Verifying component-based applications is much more difficult than verifying traditional applications because of the large size of component-based applications and the lack of information on components. Normally, the verification has to rely on the information provided by the component specification. Since our approach targets on components as its application, we have devoted a significant amount of work to developing supports for property analysis of components.

The third area of CBSE touched by our work is component evolution, especially the analysis of properties after component evolution. As aspects allow the modification of program execution, component evolution can be realised by aspects naturally. However, any modifications to components may affect the stability of component composition and even the functionalities of other components. Studies on component evolution often aim at solving conflicts between components arising from modifications to components. In this thesis, we have studied the effects of evolution on components by aspects written in our designed aspect language.

In this chapter, we review the most important concepts and approaches in CBSE that are strongly related to the our work. The remainder of this chapter is organized as follows. Section 2.2 introduces the definition and properties of a software component. Section 2.3 presents an overview of a few representative component models that have been introduced by academics as well as industry. Interaction protocols, a special type of specification often used to express communications between components, play an important role to ensure the correctness of the composition of components. Section 2.4 will focus on interaction protocols, how they are specified in general and in components, and how they can be used to reason about the properties of components and their composition. Section 2.5 presents a brief introduction to architecture description languages. Section 2.6 presents issues and major approaches for the verification of component-based applications. Section 2.7 reviews studies on component evolution and section 2.8 concludes the chapter.

2.2 Software component

This section describes software components, the core elements of component-based software. In section 2.2.1, we present one commonly accepted definition of a software component. Then we explain the critical role of the interface of software components in section 2.2.2.

2.2.1 Definition

There have been various proposals, e.g. [100, 30, 111, 61, 121] for the definition of the concept *software component* since the idea of CBSE was first floated. Among different proposals, the definition of a software component formulated by Clement Szyperski [121], which is as follows:

> *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 composition by third parties.*

is most popular and commonly accepted. This definition captures two important characteristics of a software component: self-containment with well-defined interface and subject to composition.

As a software component is a unit of composition and supposed to be deployed independently, its features should be well-encapsulated. All information on how it can be used is exposed through its interface only. These characteristics enable reusabibility of software components and thus help to achieve the goals to improve reusability, maintainability, and adaptability of component-based applications.

2.2.2 Component interface

Normally, a component-based system is built by the composition of several components which could be developed independently by different parties that ignore one another's requirements and implementation. Such composition is only possible thanks to the availability of well-defined component interfaces which should serve as contracts between the provider and the clients of a software component. On the one hand, the provider must declare the services that the component offers and the set of requirements on how those services should be used as parts of the component interface. On the other hand, clients of a component must consult the component interface to learn what services they can use and what requirements they must fulfill before using those services. Furthermore, since the interface of a component must reflect its content, it may be the only source of information about the component that can be used, e.g, to test or analyze its effects by other parties.

Let us consider a simple example of one component representing a bidder in an auction system. Figure 2.1 shows the interface of the component. According to the interface, the component provides one service called `bid` and requires two services `register, unregister` to implement the services it provides.

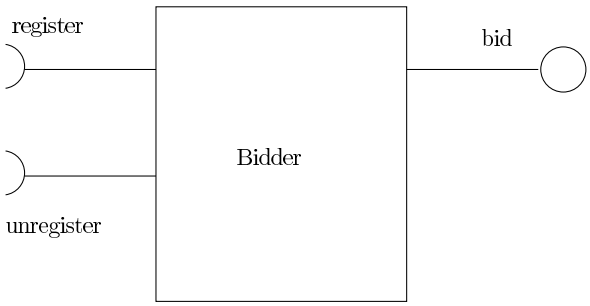


Figure 2.1: Interface of component `Bidder`

Services are typically further specified in an interface by their signatures as follows:

```
component Bidder{
    requires:
        void register(int id, int item);
        void unregister(int id, int item);
    provides:
        void bid(int item, double price};
}
```

Note that this example just shows the simplest type of a component interface which contains only signatures of services offered and required by a component. This type of interface mainly exposes the syntactical information of the component.

In principle, the interface for components should define services on the right abstraction level for the component. If an interface reveals much more details than needed, the component would be more difficult to understand for their clients and hinder future enhancements, for instance, when the component needs to be replaced but there are other components that rely on its internal implementation (which should never be exposed). On the other hand, if an interface does not carry sufficient details about a component, component use or composition may be difficult or even impossible due to mis-assumptions about the operation and requirements of that component. For instance, the above component might require that the `bid` service can only execute after the execution the `register` service and before the execution of the `unregister` service. Such a requirement cannot be explicitly specified in the component interface should the interface is designed to carry only service signatures.

Many studies on the subject about component interface suggest that signature-based interfaces are not sufficient for components since they do not provide much detail about the behavior of a component. In order to overcome that problem, interaction protocols, which allow components to specify sequences of events or method invocations, have been proposed to be included into component interface first by Daniel M. Yellin and Robert E. Strom [129] and then others, e.g, [104, 102, 106]). We believe that integrating interaction protocols into the component interface is essential for the success of component composition. Therefore, our work in this thesis relies on components that feature explicit protocols in their interfaces.

In summary, software components can be developed independently by different parties to be assembled to build larger applications. They expose their details in an abstraction manner through interfaces which describe the contracts between providers and clients of the components.

2.3 Component Models

A component model describes a set of standards for the realisation of components and component-based applications. Concretely, it defines what components are, how they can be constructed, how they can be deployed and how they can be executed.

Along with the emerging of the CBSE discipline, a large number of component models have been introduced over recent years both by the industry and the academia. Current component models can be classified into two categories: academic component models and industrial component models. Academic component models have been proposed with focus on the examination of new concepts, the formulation of properties of components, and the implementation of analysis and verification techniques for component-based applications. Industrial component models have been developed to bring new technologies to the software industry for the production of components and large-scale component-based systems.

In the following we will discuss a few representative component models developed by the academia and the industry. For each component model, we will focus on the structure of a component and how they are specified in those models. We are particularly interested in what kinds of information are exposed by a component through its interface and how far analysis of the component interface could go.

We will demonstrate the features of those component models using the example of an auc-

tion system adapted from [71]. Basically, this system involves four parties: the auctioneer, the bidder, the seller, and the item. The auctioneer plays the role of a mediator who communicates with both the bidder and the seller. The bidder has to register to the auctioneer when he wishes to participate in an auction session. He will send his bids for an item to the auctioneer and will also receive the answer from the auctioneer. The seller advertises his item through the auctioneer and also notifies the auctioneer when he decides to accept a bid and sell his item.

2.3.1 Academic models

Currently, there have been numerous component models proposed by research groups around the world, *e.g.,* SOFA [103], Fractal [31], CGEN [112], ComponentJ [16], Java Layers [33], Koala [124], etc. In this section we will discuss three models in details: SOFA, Fractal, and CGEN. The SOFA component model is among the few existing component models that support the specification and analysis of behavioral protocols in components. The Fractal component model is distinguished from other component models in its openess which provides the designers with capabilities to extend and adapt the component structure to fit their needs. The CGEN component model provides a general purpose component framework in Java with support for first-class generic types.

2.3.1.1 SOFA

SOFA (SOFtware Appliances) was introduced by Plásil et al in [103] to provide a platform for the specification and implementation of software components. An application is viewed as a hierarchy of nested components in SOFA. In the following we give a brief presentation on how components are defined and how they evolve during their life cycle according to the SOFA model.

A SOFA component is either primitive, *i.e.,* it does not contain any subcomponents, or composed, *i.e.,* it is composed from other components. It is described by its *frame* and *architecture*. The *frame* defines *provides-interfaces* and *requires-interfaces* of the component and optional properties for parametrizing the component. The *architecture* describes the structure of the component including its direct subcomponents and interconnections between those subcomponents. Interconnections between subcomponents are specified via four kinds of interface ties: binding, delegating, subsuming, and exempting. Connecting via binding would link a requires-interface to a provides-interface. Connecting via delegating would link a provides-interface of the component to a provides-interface of a subcomponent. Connecting via subsuming would link a requires-interface of a subcomponent to a requires-interface of its parent component. Finally, an exempting tie would release an interface of a subcomponent from any ties.

SOFA components are defined using the SOFA Component Description Language (CDL). The following code give examples of the CDL definitions of the *Auctioneer*, *Bidder* and *AuctioneerBidder* components. Two interfaces *IBidding* and *IRegistration* are defined to represent the set of services that provided or required by components. Three frames describe the provides-interfaces and requires-interfaces of the three components. For instance, the *Auctioneer* component implements the *IRegistration* interface so it will provide at least two method calls *register* and *unregister*. The *BidderAuctioneer* component is compposed of the *Auctioneer* component and the *Bidder* component. Its frame indicates that it provides all

the methods presented in two interfaces *IBidding* and *IRegistration*. Its internal structure is specified via the CDL architecture construct. Line 36-37 show how two subcomponents are instantiated. Line 38-39 show how the interfaces of its subcomponents are tied. Line 40-41 show how to link its provides-interface to the subcomponents provides-interfaces.

```
1   interface IBidding{
        void youGotIt(string item);
3       void youLostIt(string item);
    }
5
    interface IRegistration{
7       void register(string name,
                string item);
9       void unregister(string name,
                string item);
11  }

13  frame Auctioneer{
        provides:
15          IRegistration reg;
        requires:
17          IBidding bid;
    }
19

21
    frame Bidder{
23      provides:
            IBidding bidding;
25      requires:
            IRegistration registering;
27  }

29  frame BidderAuctioneer{
        provides:
31          IBidding BidderRegister;
            IRegistration BiddingNotification;
33  }

35  architecture BidderAuctioneer{
        inst Bidder B;
37      inst Auctioneer A;
        bind B:registering to A:reg;
39      bind A:bid to B:bidding;
        delegate BidderRegister to A:reg;
41      delegate BiddingNotification to B:bidding;
    }
```

The SOFA component model offers supports for the specification of behavior protocols to formalise the communication between two components [104]. A behavior protocol describes a trace (or sequence) of event tokens. Each event token represents one of the four communication methods: emitting a method call, accepting a method call, emitting a return from a method call, and accepting a return from a method call. Such behavior protocols can be described as a part of the component interface, either in the interface definition or in the frame definition. The following code gives an example of how behavior protocols can be described in the frame of a component. The protocol (defined with keyword `protocol`) expresses sequencing constraint on four methods *register, youGotIt, youLostIt, unregister*. That is, *register* should be called first, then *youGotIt* or *youLostIt* can be called. These calls can occur repeatedly (denoted by the repetition operator *). Finally a call to *unregister* ends the bidding process.

```
    frame BidderAuctioneer{
2       provides:
            IBidding BidderRegister;
4           IRegistration BiddingNotification;
        protocol:
6           ?BidderRegister.register; (?BiddingNotification.youGotIt +
                ?BiddingNotification.youLostIt)*; ? BidderRegister.unregister;
8   }
```

In SOFA, behavior protocols are described as regular expressions. It is possible to perform analysis on compatibility and susbtitutability of SOFA components thanks to the information

provided by their behavior protocols.

2.3.1.2 Fractal

Fractal [31] was devised initially by France Telecom and INRIA and later distributed by the ObjectWeb consortium since 2002. It provides a framework for the implementation, deployment, and management of complex component-based systems. Important characteristics of Fractal include reflexive, open, and extensible. Julia [42] is the reference implementation of the Fractal component model.

The Fractal component model is distinguished from other component models in that it offers several levels of control capabilities that a Fractal component can elect to offer to other components. For instance, at the lowest level of control capability, a Fractal component is a runtime entity, called a *base* component. The only way it can be used is by invoking operations on its component interface. In Fractal, a component *interface* is an access point to a component, which is similar to the so-called *port* in other component models. There are two kinds of component interfaces: *server interfaces* representing incoming operation invocations and *client interfaces* representing outgoing operation invocations. At the next level of control capability, a component can provide a standard interface that exposes all its external (client and server) interfaces. At the next level of control capability, a component can expose (part of) its internal structure.

Internally, a Fractal component comprises of two parts: a *controller* (also called *membrane*) and a *content*. The content of a component consists of a finite set of subcomponents that are under the control of the controller of the enclosing component. The controller of a component provides *control interfaces* to introspect and reconfigure the component's internal features. More precisely, it can intercept incoming and outgoing invocations to and from the component's subcomponents or even add behavior to the handling of those invocations. It can also suspend, do checkpointing, and resume activities of the subcomponents. It can provide an explicit and causally connected representation of the subcomponents. Note that Fractal model allows arbitrary classes of controllers to be defined so that programmers may provide their own controller classes which suit their objectives.

Figure 2.2 illustrates the Fractal composite component namely *Bidder-Auctioneer*. This component is composed of two primitive components: *Bidder* and *Auctioneer*. All three components expose their external interfaces so that they can interact with each other. Bindings between *Bidder* and *Auctioneer*, denoted by lines connecting two interfaces, represent the fact that *Bidder* uses some services provided by *Auctioneer* and vice versa. Interfaces are marked with their types: C represents all external interfaces, BC represents binding controllers, CC represents content controllers, LC represent life cycle controllers. Binding controllers allow for binding and unbinding client interfaces of components to other components. Content controllers expose internal structures of composite components and allow for adding or removing their subcomponents. For instance, *Bidder* and *Auctioneer* components do not implement content controllers so they do not expose their contents. The *Bidder-Auctioneer* composite component implements a content controller so we can see its internal structure which consists of two primitive components. Life cycle controllers allow for starting, stopping (and more) a component properly.

Communication between Fractal components is established via binding between component interfaces. Two binding types are supported: *primitive binding* which is binding between one client interface and one server interface, and *composite binding* which is a communica-

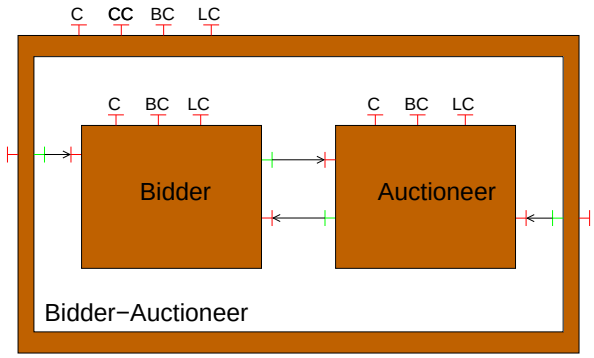


Figure 2.2: A composite Fractal component

tion path between several component interfaces. Binding can also be reconfigured at runtime through binding controller interfaces.

Fractal does not provide explicit support for the specification of interaction protocols. However, there have been on-going studies that aim at supporting the specification of behavior protocols in Fractal, *e.g.,* [75]. Furthermore, a checker called Fractal Behavior Protocol Checker (Fractal BPC) [41] is currently developed to provide a behavior protocols platform for Fractal components.

2.3.1.3 CGEN

CGEN (Component NEXTGEN)[112] offers supports for the implementation of component-based systems using Java language. As an extension of NEXTGEN [34], CGEN has an important characteristic: it supports first-class generic types, *i.e.,* generic types can be used wherever conventional types can appear, not only in class definitions but also in component definitions. The CGEN framework is built on Sun Java 5.0. A CGEN component is backwards compatible with existing libraries and can be executed on Java Virtual Machines.

CGEN components are called *modules*. A module is a generalised Java package, *i.e.,* it also contains a set of classes and has a name qualifier. However, unlike a traditional Java package, a CGEN module can specify the functionality that it imports and the functionality that it exports through its *signatures*. In other words, signatures serve as component interfaces in CGEN. CGEN language supports binding an identifier to a module instantiation.

A signature specifies the classes and their members in prototype form as in the following syntax:

```
signature S<V implements E,..., V implements E> [extends E,...,E];
sigMember*
```

where S is the name of the signature; each V is a module parameter; each E is a signature instantiation; and each $sigMember$ is either a prototype of a class, a binding statement, or an import statement. The following code shows the signature of the two modules representing an item and an auctioneer:

```
1   signature SItem;
    class Item{
3       Item();
        double getMinPrice();
```

```
5     }

7     signature SAuctioneer<A implements SItem>;
      class Acutioneer{
9         Auctioneer();
          void sell (A.Item item);
11    }
```

The signature *SAuctioneer* shows that an auctioneer module imports a module A implementing the signature *SItem*. Any module implementing the *SAuctioneer* signature must at least provide the implementation of the constructor *Auctioneer* and method *sell* with parameter of type $Item$ in module A.

A module contains a set of classes (and interfaces), especially those that are specified in its signatures. A module is defined using the following syntax:

```
module M<V implements E,..., V implements E> implements S,...,S
moduleMember*
```

where M is the name of the module; each V is a module parameter; each E, S is a signature instantiation; and each $moduleMember$ is either a class definition, an import statement, or a binding statement. The following code shows the definition of module called $MAuctioneer$:

```
1     module MAuctioneer<A implements SItem> implements SAuctioneer<A>;

3     public class Auctioneer{
          Auctioneer(){...};
5         void sell(A.Item item){...};
      }
```

As described above, a CGEN component can specify in its interface its provided services and its required services through its signatures. However, there is no support that enables a CGEN component to explicitly declare interaction protocols between the component and other components. Therefore compatibility relation in CGEN is established based on module signatures only.

2.3.2 Industrial models

There have been many successful component models coming from the industry, *e.g.,*. JavaBeans [92], Enterprise JavaBeans [96], COM [84], .NET [84], CCM [61], Web services [19]. Those component models have realised and validated CBSE principles. Furthermore, thanks to lessons learned from those models, more and more efficient component models can be developed. In the following we will discuss two major industrial component models: Enterprise JavaBeans (EJB) [89] and Microsoft.NET [84].

2.3.2.1 Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) [96] has been developed to be a server-side component architecture that supports the construction of enterprise applications. In EJB, a component is an enterprise bean hosted in an EJB container. This container is responsible for the management and execution of the bean. There are three types of enterprise beans: *entity bean*, *session bean*, and *message-driven bean*. Entity beans typically represent business objects and

are stored in a relational database. Session beans are mainly used to represent business processes. Message-driven beans listen asynchronously for messages from clients or other beans and perform tasks when receiving such those messages.

In EJB 3.0, EJB components (beans) may expose their functionalities through their interfaces. An interface consists of methods in prototype form and fields. The following code shows an excerpt of the interface of the *Item* component:

```
// interface of Item component
public interface Item{
    public double getMinPrice();
    public string getDescription();
    // other methods
    ...
}
```

The interface of the *Item* component shows that the component offers two services $getMinPrice$ and *getDescription*. Actually, the interface of an EJB component looks very much like standard Java interface. Composition in EJB is simply made by having one bean to invoke the method exposed in the interface of another bean. For example, the *Bidder* component may have a method call to *getDescription* provided by the *Item* component.

As described above, the interface of an EJB component only specifies the services it provides. Programmers as well as the compilers/intepreters rely mainly on this type of interface for deciding whether two EJB components can be composed. No sequencing constraints on using the services of an EJB component are explicitly declared in any kind of interfaces. As a consequence, such details have to be stored in the documentation and no automatic checking can be done for those constraints.

2.3.2.2 Microsoft .NET

Microsoft .NET component model [84] has been supported by the .NET framework which allows the implementation of .NET components in any .NET languages, including C#, VB, C++. Basically, source code written in .NET languages are compiled in to a form of bytecode called Common Intermediate Language (CIL). The .NET framework contains the Common Language Runtime (CLR) that gives an execution environment for the CIL code. This mechanism supports cross-language interoperability.

In Microsoft.NET, a component comprises two parts: the metadata and the implementation in CIL. The metadata, a binary piece of information automatically generated by the compiler, has been intended to provide declarative information of .NET components. Hence, metadata can be considered the interface of a .NET component. The metadata of a .NET component describes three kinds of information: (i) description of assembly (*e.g.,* identity, files, permissions to run), (ii) description of types (*e.g.,* name, visibility, base class, interfaces implemented, and members), and (iii) attributes (*e.g.,* additional descriptive elements that modify types and members).

The interface of a .NET component does not specify any interaction protocols. However, programmers may declare additional information about their components by writing customised attributes which will be added to the metadata. Interaction protocols could be informally presented in the interface that way. Anyhow, no support for compatibility checking which takes into account interaction protocols can be done.

2.3.3 Summary

We have presented a set of academic and industrial component models. We have shown how components are presented through their interfaces. As we have noted, in most of the existing component models, component interfaces mainly consist of services (and modules) that provided or required by the components. Among the component models we have described in this section, only SOFA explicitly supports the specification of interaction protocols described as regular expressions. We believe that formally specifying interaction protocols in the interface of a component would make it better understood by other components. As a result, component composition would be easier.

2.4 Interaction Protocols

An interaction protocol, in general, describes constraint on the order of actions that should occur among two or more participants in a communication. Interaction protocols make explicit important relation constraints in communications which could otherwise be hidden in informal documentation. Furthermore, they provide additional information that can be used in the verification process to ensure an implementation conforms to its protocols. Interaction protocols have been used as a means of specification in a variety of domains such as objects models ([123, 98, 114]), architecture description languages ([58, 85]), component-based applications ([104, 102, 129, 106]), agent-based applications [4], etc.

In the following we discuss different aspects related to interaction protocols. First, we present in section 2.4.1 an overview on process algebras, a family of formalisms that have been frequently used as a basis for the description of interaction protocols. Next, in section 2.4.2, we go through a few important approaches that consider using protocols to specify objects. Finally we present studies on the use of interaction protocols in the specification of component interface in section 2.4.3.

2.4.1 Process Algebras

In computer science, the process algebra (or process calculi) represent a family of algebraic approaches to the study of concurrent systems. More concretely, process algebra provide tools for the specification of processes, especially the interactions, communications, and synchronisations between those processes. Besides, they provide supports for formal reasoning about processes and systems through algebraic laws.

Numerous approaches of this type have been developed since the early eighties, for example, the Calculus of Communicating System (CCS) introduced by Robin Milner [94], Communicating Sequential Processes (CSP) [32] by Tony Hoare, Pi-calculus[95] by Robin Milner, Joachim Parrow and David Walker as successor of the process calculus CCS, Language Of Temporal Ordering Specification (LOTOS)[29] by E. Brinksma, G. Scollo and C. Steenbergen, Calculus of Broadcasting Systems (CBS)[105] by K.V.S. Prasad, etc. In the following, we give a more detailed presentation CCS and π-calculus, two of the most popular process algebras for modelling concurrent and distributed systems.

2.4.1.1 Calculus of Communicating Systems (CCS)

The Calculus of Communicating Systems (CCS)[94] was introduced by Robin Milner to model processes and communications between thems in concurrent systems. It has been also the

basis for the development of a few later process algebras such as Pi-calculus[95], LOTOS[29], and CBS[105].

Processes are central elements in the CCS model. They can be constructed using three main operators: the prefix operator (.), the parallel composition operator (|), and the choice operator (+). A process roughly corresponds to a transition system that evolves based on a set of actions fired during the execution of the process. Furthermore, two processes can communicate through synchronised actions. Lets consider the following processes defined in the CCS:

$$Server = login.logout.Server$$
$$Client = \overline{login}.(\overline{logout}.Client + update.logout.Client)$$

Assume that *Client* and *Server* are two parallel processes in the system where they can synchronise on two actions *login*, *logout* (*login* synchronises with $\overline{login}$ and *logout* synchronises with $\overline{logout}$. We can write $Client \mid Server$ where | is the parallel composition operator to model the composition of the two processes. Each process can evolve regardless of the other as long as they are not involved in the two common actions `login` and `logout`. The definition of the *Server* process indicates that the process starts with the *login* action followed by the *logout* action and then loops back to the beginning. The reference to the *Server* process itself represents recursion. The *Client* is a little more complex. It starts with the $\overline{login}$ action which synchronises with the *login* action of the *Server* process. Then it can evolve either by the *logout* action (then back to the beginning) or by a local action called *update* (then *logout* and back to the beginning). The choice operator (+) represents the ability of a process to choose one in two or more actions to process.

2.4.1.2 Pi-calculus

The pi-calculus (or π-calculus) [95] was invented by Robin Milner, Joachim Parrow and David Walker based on the CCS to describe concurrent and distributed systems in which processes may change their structures thanks to the ability to pass channels as data along other channels. A system specified in pi-calculus may consist of a set of parallel processes like in the CCS. These processes can be constructed using CCS-like operators such as the prefix operator (.) to describe sequences, the parallel composition operator (|) to describe concurrency, the choice operator (+) or by recursion definition to describe repetitions. The pi-calculus extends the CCS with more constructs to describe dynamic communication, encapsulation, and repetition. Lets consider an example to demonstrate these new features. Suppose we need to model a procedure call between a client and a server. The server runs a function that simply takes an identifier from a client and then returns the data item corresponding to that identifier to the client. Hence, the server can be modeled as a process as follows:

$$Server =! \; get(c, id) \; . \; \overline{c}(d)$$

That is, the Server process waits for a message that was sent on channel *get* accepting two inputs: c (the name of the channel the server will return the data item) and id (the identifier of the data). After receiving a message on *get* channel, the process proceeds to send back the data item d on channel c which it has received from the client. The replication operator (!) is used to indicate that the server process can always create new copies of itself to serve several clients.

The client process, which obviously consists of a request to the server's provided service, can be modeled as follows:

$$Client = (\nu c)(\overline{get}\langle c, 5\rangle | c(t))$$

That is, the Client process consists of two parallel sub-processes: (i) the $\overline{get}\langle c, 5\rangle$ process that sends on channel *get* the channel c to get back the reply and the identifier '5', (ii) the $c(t)$ process that waits on channel c for a result that will be bound to t. The ν operator is used to allocate a new channel within each client that sends request to the server.

Both the CCS and the pi-calculus are Turing-complete. As a consequence, they are powerful as modeling languages. However, since they are very general, many fundamental analysis problems are undecidable for these languages.

2.4.2 Interaction protocols in object models

In the following we discuss two representative approaches for the specification of interaction protocols in object models. The first approach is based on Nierstrasz's landmark work on "regular types for active objects" [98]. The second approach is based on the notion of "session types" introduced by Kohei Honda [66]

2.4.2.1 Regular types for active objects

One of the first approaches to equipping objects with protocols was introduced by Oscar Nierstrasz through his work on "Regular types for active objects"[98]. In that work, an object is viewed as a process defined in the process algebras such as the CCS and pi-calculus introduced above. Concretely, every object provides its services by accepting certain types of requests along its named channels (one per request name) and in turn, emitting a certain type of reply in response to a request. Moreover, the object does not have to accept a particular request all the time, it may enable the request (and thus make the corresponding service available) depending on the current state it is at. Hence, in addition to types of provided services, the specification of an object also includes a protocol that formulates the sequences of requests that the object can serve. In Nierstrasz's object model, object protocols are backed by finite state machines and thus the types of such objects are called regular types.

Two important notions have been studied for regular types: substitutability and sastifiability (or compatibility). Since an object is viewed as a process, it can be modeled as a transition system where a transition represents a request being received.

Let $traces(x)$ be the set of sequences of requests that an object in state x can accept:

$$traces(x) = \{s \mid \exists x', x \stackrel{s}{\Rightarrow} x'\}$$

In the above definition, s is a sequence of requests and x' is the state into which the object leads after accepting sequence s from state x.

Let $init(x)$ be the set of requests which are initially enabled from state x, *i.e.*,,

$$init(x) = \{r \mid \exists x', x \stackrel{r}{\rightarrow} x'\}$$

where r is a request.

Let $failures(x)$, the set of failures of an object in state x, be defined as follows:

$$failures(x) = \{(s, R) \mid \exists x', x \stackrel{s}{\Rightarrow} x', R \text{ is finite }, R \cap init(x') = \emptyset\}$$

where s is a sequence of request that makes the object evolve from state x to state x' and R is the set of requests that are not accepted in state x'.

In principle, if an object a conforms to the protocol of another object b then a can safely substitute b since the clients of object b can make the same sequences of requests that they expect to be served by b to object a. Hence, the substitutability relation on object protocols is formulated as follows:

> "An object in state x is *request substitutable* for an object in state y, written $x :< y$ iff
>
> $$(i) traces(y) \subseteq traces(x)$$
> $$(ii) failures_y(x) \subseteq failures(y)$$"

where $failures_y(x) = \{(s, R) \in failures(x) \mid s \in traces(y)\}$.

The first condition requires that the object in state x must accept at least the same set of sequences of requests that are accepted by another object in state y so that x can satisfy the same clients. The second condition establishes that if the object in state x, after accepting those same sequence of requests, refuses a request r then that same request r must also be refused by the original object in state y. This requirement ensures that x cannot refuse more request than y after accepting the same sequences of requests. Since object protocols are of regular types and represented by finite state machines, there exists a simple algorithm based on checking equivalence of finite state machines to check the two above conditions and thus checking for substitutability relation object protocols are decidable. This is the main advantage of this object models comparing with process algebras such as CCS, pi-calculus which are much more general.

The notion of compatibility on object protocols in Nierstrasz's object model is established for an object which provides services by accepting requests (a server) and an object which requests for those services (a client). Informally, two object protocols are compatible if the client object can always make progress when two object interacts. In other words, the server object can terminate only if the client will not send any further request. This compatibility relation is formalized as follows.

Let $offers(c) = \{(s, R) \mid \exists c', c \stackrel{s}{\Rightarrow} c', R = init(c')\}$ be the set of offers of a client c. That is, (s, R) is an offer of c if c may issue the sequence of request s and then the set of requests R. The protocols of object x and c are compatible iff

$$(s, R) \in failures(x) \cap offers(c) \Rightarrow R = \emptyset$$

That is, x can always satisfy at least a request from c after accepting any sequence of requests s from c. Since object protocols are of regular types, their compatibility relation can be determined by a simple algorithm based on the calculation of the product of two finite state machines. Furthermore, it has been proved that if an object x is request substitutable for an object y and y and c are compatible then x and c are also compatible if we only consider the behaviors of x that can be justified as acceptable by y.

2.4.2.2 Session types

There have been a number of studies, *e.g.,* [46, 43, 68], on the definition of object protocols developed based on the work on *session types* first introduced by Kohei Honda [66]. These

studies aim at the formulation of a type theory for communications between processes. Such type theory would provide a high-level abstraction for conversations in communication-based applications. Informally, session types describe protocols of conversations conducted over private channels. A session type definition exposes the structure of a conversation and types of messages exchanged during the conversation. For example, the session type '! *int* . ? *bool* . *end*' indicates a conversation in which an integer value will be send then a boolean value is received and finally the session terminates. Hence, a conversation involved multiple interactions now can be logically represented by a session type. As a result, communication safety could be guaranteed through type checking techniques developed for session types.

Hu et al. [68] have put forward the first full implementation of session types in SessionJ (or SJ), an extension of Java to support session types. Lets consider an example for a demonstration of how session types are defined in SJ. In this example, two session types representing two sides of a simple conversation between a buyer and a seller are presented. The following code shows the definitions of these types in SJ.

```
1   protocol buyItem{
        begin.
3       ![
            !<String>.
5           ?<String>.
            ?<Double>
7       ]*
        !{
9           BUY: !<CreditCard>.
                ?<Receipt>,
11          QUIT:
        }
13  }
```

```
1   protocol sellItem{
        begin.
3       ?[
            ?<String>.
5           !<String>.
            !<Double>
7       ]*
        ?{
9           BUY: ?<CreditCard>.
                !<Receipt>,
11          QUIT:
        }
13  }
```

As shown by the code above, the session type representing the buyer side of the conversation is defined by protocol *buyItem* and that of the seller side is defined by protocol *sellItem*. Note that the '!' symbol indicates the side that initiates a communication and the '?' indicates the side that follows the corresponding communication. According these two protocols, the buyer initiates the conversation by sending the description of type *String* (line 4) of an item she is looking for. The seller replies with the item's description of type *String* and its price of type *Double* (line 5-6). The buyer may ask again and again (marked by repetition symbol '*' on line 7) for another one until she comes to a decision. Then she has two options: she may buy the item by sending her credit card details and receive a receipt (line 9-10) or she may decide not to buy any item (line 11). Such options are expressed by the selection construct '{BUY,QUIT}'. The seller will act accordingly. The session (conversation) then terminates.

Building a communication-based application in SJ consists of two steps. The first step is the specification of the protocols of the conversations that may occur. The second step is the implementation of these protocols using session operations supported by SJ. The following code shows an excerpt of the implementation of protocol *buyItem*.

```
1   // s_buyer is the session socket at the buyer side for the conversation
    s_buyer.request(); // the buyer requests a session with the seller
3
```

```
    boolean decided = false;
5
    s_buyer.outwhile(!decided){
7       s_buyer.send(item_description);   // !<String>
        String item_to_buy = s_buyer.receive();  // ?<String>
9       Double price = s_buy.receive();   // ?<Double>
        if(... // want to make decision)
11          decided = true;
        else
13          ... // modify item description
    }
15
    if(... // want to buy the item){
17      s_buyer.outbranch(BUY){
            s_buyer.send(credit_card);
19          s_buy.receive(receipt);
        }
21  }
    else{ // want to quit
23      s_buyer.outbranch(QUIT){...}
    }
```

The implementation of the buyer (and the seller) should conform to its declared protocol. In the above implementation, the buy initiates the conversation by requesting a session with the seller (line 2). The code from line 6 to line 14 of the implementation reflects the definition from line 3 to line 7 in protocol *buyItem*. Construct `outwhile` (and its counterpart `inwhile`) is used to implement the repetition construct '![...]*' (and '?[...]*' respectively) in the protocol definition. Communications involved exchanges of item descriptions and prices are implemented using `send` and `receive` methods (line 7-9). The selection construct '{...}' in the protocol definition is implemented using `outbranch` (for '!{...}) and `inbranch` (for '{...}') methods. For instance, the code from line 16 to line 23 shows the implementation of the part of the *buyItem* protocol in which the buyer makes decision on whether to buy the item or just quit the session. There would also be a similar implementation that conforms to the *sellItem* provided for the seller.

Communication safety is guaranteed through SJ supports for type checking for session types. At the local level, static type checking is performed to ensure that the implementation conforms to its declared protocol. For example, the above implementation of the buyer will be checked against the *buyItem* protocol. Then at runtime, when two sides of a conversation are about to start a session, they exchange their session types and independently validate the compatibility of their protocols. If their protocols are compatible, a session will be established for further communications. In other words, both sides will know the structure of their conversation through the declared protocol of each other and these protocols must be compatible before any actual communications can be made.

Session subtyping is realised in SJ in two forms. The first form of session subtyping permits message type variance following the subype principle of object-oriented programming for `send` and `receive` operations. That is, a subtype can be sent wherever a supertype is expected. The second form of session subtyping permits structural subtyping [54] for branching and selection. An outbranch implementation could select a subset of options (not all) declared in the protocol. Besides, at runtime, two protocols are compatible if the server side (inbranch) supports a superset of cases that the client side (outbranch) may require.

2.4.3 Interaction protocols in component models

In this section we present studies on the integration and application of interaction protocols in component models. We will start with Yellin and Strom's work [129] on the augmentation of component interfaces with interaction protocols. This work has been the basis for several later studies on the application of protocols in components. Next, we discuss a few other approaches on this subject covering both regular and non-regular protocols.

In [129], Yellin and Strom argue that sequencing constraints on the order in which messages are sent should be documented in the formal interface specification of a component in addition to method signatures. Sequencing constraints are defined by finite state automata whose transitions represent messages being sent or received from a particular state. A protocol describes the interaction between exact two components. For example, the following protocol describes the protocol of a server component that interacts with its clients by geting requests for data and sending back data in response to clients'requests:

```
Protocol{
    States { Init, Response};
    Transition {
        Init: +get --> Response;
        Response: -send --> Init;
    };
};
```

The above protocol is modeled by a finite state machine consisting of two states and two transitions. The server begins in state *Init* and waits for an incoming message represented by transition $+get$. Then it enters state *Response* and sends back the data in request. This event is modeled by the transition $-send$ representing an outgoing message.

A notion of protocol compatibility was established so that composability between interacting components is statically checkable. Informally, two protocols P_1, P_2 are compatible iff two following conditions hold: (i) they can always collaborate, for example, if P_1 is in a state where it can send m then P_2 will be in a state where it can receive that message m, (ii) communication between P_1, P_2 are deadlock free. Since protocols are modeled by finite state machines, protocol compatibility problem is decidable. An algorithm for checking protocol compatibility was presented in [129]. The above approach is more general than earlier approaches on object protocols in that component protocols describe both provided and required services, and either components in a communication can initiate messages rather than just the clients. However, multi-party communications are not supported and regular protocols could be insufficient for many concurrent and distributed systems.

Alfaro and Henzinger introduced Interface Automata[44], another automata-based language to specify protocols for component interfaces. The notion of compatibility established in this work is distinguished from other studies in that it is based on a so-called opstimistic view: two components are compatible if "there is some environments that can make them work together". This approach supports an efficient algorithm to automatically check for compatibility based on the calculation of the product automaton of two interface models.

While there have been numerous studies on regular protocols for components, there exists very few approaches that consider non-regular protocols mainly because non-regular protocols normally cannot provide much support for analysis. Anyhow, there have been still a

few studies that introduce different kinds of non-regular protocols to components in some particular application domains.

Ralf Reussner proposed to use counter-constrained finite state machines [107] to describe component protocols. Basically, a counter-constrained finite state machine is composed of a traditional finite state machine and a finite set of counter-constraints which can be used to impose restrictions on the occurences of services that involve with shared resources. This approach allows static checking of protocol compatibility and substitutability thanks to the fact that inclusion and equivalence problems are decidable for counter-constrained finite state machines.

Another approach in which non-regular protocols are integrated to component interfaces has been proposed by Mario Südholt [118]. In this work, a protocol language is defined as an extension of the aforementioned non-regular process types introduced by Puntigam. Therefore, the protocol language can benefit from the analysis supports of non-regular process types for protocol compatibility and substitutability.

2.5 Architecture description languages

An architecture description language (ADL) is used to describe software architectures. It permits an abstract view on an application system in terms of components, connectors, and configurations [90]. A large number of ADLs have been developed over the years, *e.g.,* ACME [58], Rapide [85], Darwin [88], Wright [18], ArchJava [16]. In the following we present a brief overview of two ADLs: Wright and ArchJava.

The Wright language [18] is an ADL that supports the description of software systems based on *component* and *connector* types. A *component* is described by a set of *ports* and an abstract description of its behavior. *Ports* are interaction points between a component and the environment. A *connector* type is defined by a set of *roles* specifying the behaviors of entities in an communication, and a *glue* specifying how these behaviors are synchronised. In Wright, behaviors are interaction protocols defined using a variant of CSP [63] notation. The configuration of a system is described by linking component ports to connector roles. It is possible to automatically check for architectural properties such as deadlock freedom or compatibility between ports and roles using model checking technique.

There have been quite a few ADLs that support the specification of interaction protocols but they normally lack facilities to check whether an implementation conforms to such interaction protocols. Now we continue with the ArchJava [16] language, an ADL that supports checking whether an implementation respects the architecture. ArchJava is an extension of Java that unifies software architecture with implementation. New language constructs are added to Java to support the definition of architectural elements such as *components*, *connectors*, and *ports*. A *port* is used to declare a *provided* method, a *required* method, or a *broadcast* method. A *component* is a special kind of object that can ony communicate with other components through explicit declared ports. For example, if a component A wishes to call a method provided by component B, component A has to connect to the port of component B that represents its provided method. Communication integrity, which requires that components in an architecture must not directly invoke the methods of other components (except its own subcomponents) but only invoke those methods by connecting to declared ports, can be enforced by the language semantics. In other words, the ArchJava's type syste, ensures that the implementation will conform to its corresponding architecture. However,

ArchJava has not yet provided supports for the specification of interaction protocols.

2.6 Analysis of component-based systems

In this section, we consider the analysis of component-based systems. As emphasized in previous sections, component composition plays a vital role in building large systems from components. If we want to assemble two or more independent components, we have to make sure that they are smoothly composable in order to achieve a well-defined anticipated result from the composition. Therefore, compositional properties, *i.e.,* properties concerning about correct composition of components, are among the most desirable properties for component-based applications. In the following, we will cover work on the analysis of such compositional properties.

This section is organised as follows. First, we present different definitions for two of the most important compositional properties: compatibility and substitutability. We then present a few approaches that deal with the verification of compositional properties for component-based systems. Finally, we conclude by summarizing the key issues that need to considered when developing analysis techniques for component-based systems.

2.6.1 Compositional properties

Ensuring the satisfaction of compositional properties helps to avoid problems arising from mismatches of the composed components. A large number of studies,*e.g.,* [98, 54, 55, 126], have been presented on techniques to verify those compositional properties. Note that analysis tasks are conducted based on the information available about the components included in a composition. Hence, the explicit knowledge of the semantics (which is mostly exposed through a component's interface) may affect if and how properties of the system can be determined. In the following we describe the principles on which component compatibility and substitutability, two essential compositional properties that are usually of concern for component composition, can be evaluated.

2.6.1.1 Compatibility

Informally, component compatibility or interoperability is the ability of two or more components to cooperate correctly. This is the most basic but also important requirement for component composition. Compatibility properties can be roughly categorized into signature, protocol, and semantic compatibility.

Compatibility at the signature level deals with elements such as names, operations, parameter types, return values, etc. This level of compatibility is currently well-defined and supported by all of the existing component models, mostly as type compatibility on signature elements.

Compatibility at the protocol level concerns the order of message exchanges between components. Compatibility at this level is more powerful than signature compatibility because it can tell more about the behavior of the composed component.

Compatibility at the semantic level is the most powerful since it concerns the actual and complete behavior of components. However, ensuring semantic compatibility, especially for components developed by third parties, is much more difficult or even infeasible in theory or practice.

Hence, protocol compatibility is a compromise between signature compatibility and semantic compatibility. Since it does not deal with all the semantic aspects of components like at the semantic level, it is easier to realize. As previously mentioned, we have opted to base our work on components with explicit protocols. Therefore, we will treat component compatibility both at the signature level and the protocol level.

2.6.1.2 Substitutability

Like most other software systems, component-based systems need to evolve from time to time due to, for instance, functionalities to be added, new requirements, implementations that have to be corrected. In many cases, modifications to an existing system are done simply by replacing one or more original components by other ones. Although such modifications are meant to enhance the original system, they may also introduce errors or break the consistency of the original system.

Component substitutability expresses the ability of one component to be substituted by another component without causing any unexpected effects to the whole system. In addition to its importance to component evolution, the notion of component substitutability also establishes the basis for component retrieval, a process to locate and identify appropriate components. Component application assemblers may wish to look for components that can substitute a specification to use for their composed systems.

Component substitutability can also be categorized three-fold just as component compatibility. At the signature level, checking for substitutability roughly means checking whether the new component offers at least all the services offered by the original ones and requires at most all the services required by the original ones. This is similar to the notion of subtyping in object models. At the protocol level, substitutability of one protocol by another one requires the new protocol to be able to accept at least the same sequences of messages and not to reject more sequences of messages as the original ones [98]. At the semantic level, a substitutability check is even more complex and hardly practical than checking for compatibility.

2.6.2 Analysis of component composition

The issues of component compatibility and substitutability have been addressed by many researchers. In this section we discuss the following approaches: (i) checking compatibility using automata-based computation [24], (ii) checking substitutability using automata-based computation [126] and (iii) proving substitutability by construction [55]. Automata are widely used to specify component protocols so checking protocol compatibility and substitutability of two or more components often boils down to the computation on the corresponding automata. Since our work is based on visibly pushdown automata, a specific type of automata, we are interested in studies where automata are used for protocol specification and analysis. The first two approaches are among those studies. The first approach focuses on techniques to deal with state space explosion, which is a very common problem of state space exploration techniques. The second approach establishes a formal foundation for reasoning substitutability properties on different levels of component abstraction. The third approach is rather unconventional for establishing substitutability property based on specific operators that have been used to construct protocols.

Checking compatibility using automata-based computation. Attie et. al [24] have proposed a verification methodology in order to prove behavioral compatibility and temporal properties of component-based systems while trying to avoid the state space explosion problem. Usually, in order to check the compatibility of two components, we can construct the product of two automata representing the behaviors of those two components and explore the resulting automaton to reason about the composition. If there is another component involved in the composition then another product automaton is calculated for the previous product automaton and the automaton of the new component. Such product construction makes the size of the model to be checked exponential when the system consists of a large number of components. Consequently, verifying such systems by exploring the calculated product becomes infeasible due to the limitation of resources.

In their work, behaviors of components are modeled as I/O automata [86]. This approach relies on pair-wise composition of automata to avoid the state space explosion problem. It does not require the computation of the product of all behavioral automata. Only the products of pairs of automata of two directly interacting components are calculated. Properties are verified for those pair-products using a model checker and then combined to deduce global properties of the system by a suitable temporal logic deductive system. This verification method has shown to basically achieve polynomial complexity in the size of components and thus could support practical behavioral compatibility checks for large scale systems.

Checking substitutability using automata-based computation. Let us now move on to a study on component substitutability. Cerna et. al [126] provides a formal foundation for establishing component substitutability with an underlying formalism called component-interaction automata. The component interaction automata language expresses a component by labeled transition system with structured labels to capture components involved in actions and a hierarchy of component names to reflect the architecture of the component. The authors have formalised three different notions of substitutability of components: (i) equivalence (one component can be replaced by another one and vice versa), (ii) specification-implementation (one specification can be substituted by an implementation of a component, in other words, the component implementation is compliant to the specification), and (iii) substitutability of non-equivalent components (one component can be replaced by another but the other way around is not possible).

The equivalence of component-interaction automata can be defined at different levels of accuracy since an observable set X of structured labels are taken into account to determine the equivalence relation (which is essentially the bisimulation relation defined for automata). The composition of components is represented by the composition of component-interaction automata which involves the calculation of the product automaton like in other approaches. However, the composition of component-interaction automata can be refined with an additional parameter specifying the set of labels indicating actual communications between two components. This helps to ignore unnecessary synchronisations that are syntactically possible but never occur in the system. The authors have shown that if two components are equivalent with respect to the set X of all input and output communications then one component can be replaced by another one and the new composed system is still equivalent to the original one with respect to the same set X of input and output communications.

According to that work, in order for a component to be able to substitute a specification, at the interface level, it has to provide and require all the services that are provided and required

by the specification (but it can provide and require more services than the specification does). At the behavior level, the automaton of the component has to behave observably the same as the automaton of the specification on every service provided and required by the specification. On the other hand, substitutability of non-equivalent components where a new component substitutes an older component can be achieved with the specification-implementation relation but the new component does not have to behave the same as the old one if such behaviors are not used by the environment.

Proving substitutability by construction. Farías and Mario Südholt [55] have introduced an approach to ensure component substitutability by construction. The approach is designed to apply over component with explicit protocols so the substitutability of components requires protocol substitutability. A set of protocol construction operators have been investigated to see if they preserve substitutability properties. It turned out that most of the operators did not preserve substitutability in general. However, there are some possibilities that substitutability can hold if we restrict the applications of the operators to some specific cases.

2.6.3 Summary

Correct composition of components is crucial to the correctness of a component-based system. As a consequence, verification and analysis of compositional properties, including two major ones: compatibility and substitutability, play important roles in the development process. Since components are often developed by third parties, verification can be even more complicated than if it is done by the authors of components. Most of the verification work rely on the details exposed by the component interface. Beside checking for the compositional properties at the syntax level, many approaches go further to take into account the behaviors of the components which are often represented by some kind of automata and perform suitable verification on those automata. We should keep in mind that any analysis techniques should not be too complicated or too expensive in order for the reasoning to be practical.

2.7 Component evolution

Just like any other system, component-based systems must be continuously adapted to fix existing bugs and problems, to add new features, to address changing requirements, to enhance system performance, etc. Evolution on component-based systems often leads to the need for new components to be added to existing systems. Then the two main questions of interest in our context are:

1. Does the evolution introduce any problems to the system due to incompatibility between the old components and the new ones ?

2. What can we do to maintain compatibility after evolution ?

One important part of our study focuses on dealing with problems arising after evolution to components introduced by aspects. In contrast to the few studies on the effects of aspect-based evolution on components, there are many studies on dealing with component evolution in a more general context, *i.e.,* non aspect-based evolution. In this section, we will give an

overview on studies that aim at addressing the above two questions. The section is organised as follows. We will first summarise in section 2.7.1 common incompatibility problems that arise from evolution on component-based systems. Then we present in section 2.7.2 several studies that aim at ensuring compatibility between components after evolution.

2.7.1 Problems arising from evolution

When new components are introduced to an existing system, problems may arise due to incompatibilities (or mismatches) between the old components and the new one. There are many reasons that can lead to such problems. In Becker et. al [25], the authors have enumerated and classified different kinds of component mismatches to help identify incompatibility problems and provide a basis for the development of automatic adaptation techniques. In that study, incompatibilities problems are classified into three categories including signature mismatches, protocol mismatches, and quality attribute mismatches.

Signature mismatches occur at the syntax level when connecting a provided service of a component to a required service of another component. For example, the same method, or parameters, or types are named differently, the orderings of parameters are different, the numbers of parameters are different for two connecting services. Protocol mismatches occur when the two protocols of two components involved in a communication are incompatible. For instance, consider that a new component A is substituted for component B. Another component C that previously communicated with B now needs to have the same communication with the new component A. However, the new component A sends more messages to component C than B does and C does not expect to receive those extra messages. Quality mismatches involve problems with non-functional properties such as security, persistency, performance, etc.

2.7.2 Approaches to deal with evolution problems

Depending on the nature of the incompatibility, different adaptation techniques are developed to "turn" incompatible components into compatible ones. Using adaptors to mediate the interaction between incompatible components is the most common method to deal with incompatibility problems. A number of studies on this subject have been proposed over the years. However, most of them, *e.g.,* [36, 110, 27], only deal with incompatibility caused by signature mismatches. Although protocols are now widely present in specifications of components, there are still only few studies that focus on the adaptation techniques which can deal with protocol incompatibility. In the following we give brief presentations on two approaches that focus on how to automatically or semi-automatically generate adaptors to solve a limited set of protocol mismatch problems. The first one is a well-known approach which has been a reference for many further developed solutions to the evolution problem. The second approach is interesting in our context because of its (partial) similarity to our approach in solving the problem.

Yellin and Strom [129], along with their proposal to augment component interfaces with interaction protocols, introduced a technique in which adaptors are used to solve protocol incompatibility. In this approach, an adaptor is modeled as a finite state automaton interfacing two interacting components. Messages exchanged between two components are intercepted by the adaptor so that necessary "arrangements" can be made to ensure that both components can evolve smoothly. The authors have developed an algorithm to automatically generate

these adaptors based on an interface mapping. An interface mapping consists of a set of mapping rules indicating how to relate messages and parameters in the two interfaces of two components.

Schmidt and Reussner [113] shown how to semi-automatically generate adaptors to resolve synchronisation problems of concurrent components at the architectural level. In this study, the authors adopt a component model which supports the description of finite state protocols in the component interface. Three common cases where an adaptor is needed have been identified and an algorithm for generating adaptors has been developed for each case. In the first case, an adaptor is generated for three components A, B, C where A uses the services of B and C. This adaptor "combines" the interfaces of B, C into a single interface that can be consulted during a compatibility check. In the second case, an adaptor is generated to synchronise the services provided by two concurrent components. For example, an adaptor between a Consumer component and a Producer component that respectively gets and puts data from/to a buffer is defined in order to ensure that the producer always puts the data into a buffer before the consumer makes a request to get the data. The last case concerns protocol incompatibilities between two components that occur when a component A requests a method m provided by component B but component B has not yet been in a state where method m can be called. The authors propose prefixing the call to method m with a sequence of calls in order to bring component B into the appropriate state where m is enabled. An algorithm has been developed to solve the problem of finding such sequences.

Recently, there have been a body of works that aim at using aspect-oriented techniques to implement adaptors for component-based systems. We will cover those types of approaches in the next chapter.

2.8 Conclusions

In this chapter, we have discussed three important subjects on component-based software engineering, including component specification, verification, and evolution. For component specification subject, we have focused on the concepts of component interfaces, interaction protocols, and the importance of integrating interaction protocols into component interfaces. We have also presented a few typical component models and architecture description languages. On the analysis and verification subject, we have discussed two of the most important compositional properties of component-based systems, compatibility and substitutability. These properties often need to be verified in order to establish the confidence about the coherence of a component-based system. On the subject about component evolution, we have considered problems arising from evolution on components and a few approaches to deal with those problems.

These subjects are closely related to our work which is about the development of an aspect language to define aspects over components. We choose to work on the application of aspect-oriented programming (AOP) to components because crosscutting concerns abound in large scale component systems and the AOP paradigm (this subject will be presented more completely in the next chapter) is particularly suitable for handling crosscutting concerns. Based on what we have learned from previous studies on component-based software engineering, we consider a set of requirements for an aspect language dedicated towards components. First, as component models with interaction protocols become more popular, the aspect language for components should provide facilities to describe component protocols. Second, this language

should also support the analysis or verification of compositional properties such as compatibility and substitutability which are important for component-based systems. Furthermore, an analysis technique should be simple enough for it to be practical and to be done in an automatic manner. Finally, since we aim at using aspects to modify components, appropriate support is needed in order to reason about the effects of aspect-based evolution on the original component-based system.

Chapter 3

Aspect-Oriented Programming

3.1 Introduction

Nowadays, software applications are not only more powerful but also more complex due to both the advent of innovative technologies and the increasing needs and expectations for software applications. A complex application usually has to be able to handle a variety of concerns such as security, performance, distribution, and failure recovery. Although innovative technologies make it technically possible to incorporate solutions for different concerns into one software application, different concerns are often so scattered and tangled that the resulting software system becomes very complex. Such complexity makes it difficult for the understanding, maintenance, and evolution of software. Solving the problem of scattering and tangling concerns in order to reduce software complexity is the subject of the Separation of Concerns paradigm in software engineering.

Separation of Concerns (SoC) is a key software engineering principle which requires that different concerns composing in a software system must be separated into well-separated program modules. Such separation helps to reduce the complexity of software applications and thus makes software easier to write and understand. Moreover, it is much easier to provide support for software evolution if concerns that require different types of modification are well-separated. Besides, as separation of concerns typically results in loose coupling of elements of a software, it helps to increase flexibility and reusability of individual software components.

Since SoC advocates breaking software into loosely-coupled concerns, support is needed to "glue" those concerns together so that they integrate smoothly. Although the traditional object-oriented (OO) programming paradigm supports building of software from smaller entities such as classes and modules, it does not provide sufficient support for implementing the SoC principle because there are scatting and tangling concerns that cannot be modularized as class objects or modules. Aspect-oriented programming (AOP) has been introduced as a complement to OO programming paradigm for the systematic treatment of SoC. More concretely, AOP aims at the separation of crosscutting concerns, which are concerns that cannot be modularized using traditional OO techniques. The term *aspect* is used to refer to a modularization mechanism (such as a language or design level abstraction) for crosscutting concerns.

In recent years, there have been an increasingly large number of approaches to aspect-oriented (AO) technologies and aspect-oriented software development (AOSD) methods. Most

of these approaches focus on the development of semantics, language support and implementation which are essential to the feasibility and applicability of the AO technique in reality. A number of AOP languages and development tools have been introduced. There are also approaches to exploit AOP in specific application domains or industrial application systems. However, many limitations still exist which leave much room for further research.

This chapter presents a thorough review of three critical subjects on AOP that are strongly relevant to this thesis: language support for AOP, verification for AO programs, and AOP for component-based software. The chapter is organized as follows. Section 3.2 covers four major AOP approaches. Section 3.3 focuses on studies on verification and analysis techniques for AO programs. Section 3.4 describes AOP studies in one specific application domain: component-based software. Finally, section 3.5 concludes the chapter with lessons that we have learned and what features a good AO language should offer in order to be powerful for general purposes and especially for component-based systems.

3.2 AO approaches

In this section, we review four major approaches on the development of AOP technologies. We first start with AspectJ, the most popular and successful aspect-oriented language until present. Then we continue with an important group of AO languages: history-based aspect languages. These aspect languages are more advanced than earlier developed ones in that they can take into account the order of events in the declaration of aspects. The approaches introduced in this subsection are most clearly related to the work presented in this thesis. Finally, we briefly present adaptive programming and composition filters, two of the earliest AO approaches that share same principles of SoC with the current work.

3.2.1 AspectJ

AspectJ [74] is an aspect-oriented extension created at Xerox PARC for the Java language. AspectJ provides support for programmers to write aspects to modularize crosscutting concerns and a mechanism to incorporate those aspects into the main program to produce the final systems. AspectJ has become the most mature AO language that is usually refered to as the de facto standard for an AO language in the AOP research community and started getting attention from the industry. The AspectJ model has been transposed in multiple other languages including C++, Smalltalk...

In the following we first give an informal and example-based presentation of critical elements defined by AspectJ, including join point, pointcut, advice, and aspect. We then discuss some advanced issues and the limitations of this language with respect to those issues.

Let us consider a simple example of a data system where users can perform a set of operations on data: `query, add, remove, commit, modify`. Each operation is implemented as a method in the application's source code. The following piece of code presents an excerpt of the data module:

```
1 public class Data{

3     public void add(Item itm);
      public void remove(Item itm);
5     public void modify(Item itm);
      public Item query(String str);
```

```
7     public void commit(Item itm);
  }
```

Assume that we would like to execute a method called `writeLog` right after executions of five above methods. Normally, a method call to `writeLog` will be simply placed after every call to methods `query, add, remove, commit, modify` using traditional object-oriented languages. This implementation makes the code invoking `writeLog` to scatter over the system. Imagine that `writeLog` needs to be replaced by another method or to be removed, we would have to revisite all places where it is called and modify the code. AspectJ provides language facilities to better modularize crosscutting concerns such as `writeLog` by introducing a modular unit called *aspect*. In AspectJ, an aspect declaration has a similar form to a Java class declaration. It encapsulates a crosscutting concern and describes where and when some code will be executed. In principle, AspectJ features a dynamic join point model in which join points are well-defined points in the execution trace of the main program. A join point can be a method call, field assignment, handler execution,... When the main program runs, its join points are observed so that crosscutting concerns declared by aspects can be injected to the main program's execution. The following code presents the aspect implementing the logging concern:

```
    public aspect Logger {
2
        public void writeLog(){
4           System.out.println("Write to log file.");
        }
6
        pointcut dataop(): execution(* Data.add(..)) || execution(* Data.remove(..))
8                          || execution(* Data.modify(..)) || execution(* Data.commit(..))
                           || execution(* Data.query(..));
10
        after(): dataop(){
12          writeLog();
        }
14  }
```

Line 3-5 describe the implementation of the `writeLog` method that handles the logging task. Line 7-9 describe a pointcut declaration. A *pointcut* is a collection of join points and serves as a pattern to match against the program's join points at runtime in order to identify certain execution contexts. In the above example, the `dataop` pointcut captures all the join points where five methods `query, add, remove, commit, modify` are called. Line 11-13 describe the *advice* in the aspect. In AspectJ, advice is a method-like mechanism to declare certain code to be executed when its pointcut successfully matches some join points at runtime. In other words, advice defines the crosscutting behaviors. The advice in the above example specifies that `writeLog` will be executed after any join point that matches one of the join points declared by the `dataop` pointcut. The wild cards in the pointcut declaration indicate that the pointcut can match methods of all return types. Hence, AspectJ allows us to modularize a crosscutting concern into a unit in which pointcuts define conditions for the advice which defines crosscutting behaviors to be executed. Consequently, if we would like to replace the `writeLog` method or remove it, we just need to modify the logging aspect.

The implementation of any AOP language concentrates on coordinating aspect and non-aspect code so that they run correctly. This integration process is often refered to as *aspect*

weaving. Aspect weaving integrates the aspect code with the base program code in a way that aspect advice is able to run at applicable join points that are successfully matched by pointcuts. Basically, aspect weaving performed by AspectJ compiler of Eclipse (an IDE supporting AspectJ) consists of two phases. First, AspectJ code is compiled into pure Java bytecode annotated with additional attributes to indicate advice and pointcut declarations. Then the byte code of the base program is instrumented with calls to the pre-compiled advice method. The details of AspectJ's weaving process can be found in [62].

On the one hand, AspectJ has its strengths as the most mature AO technology available today. AspectJ is designed to be a seamless extension to Java so it is simple and easy to learn. Furthermore, tool supports for AspectJ are widely available, for example, plug-in for Eclipse integration. On the other hand, as one of the first-generation aspect languages, AspectJ has its limitations and potentials for further research.

One of the potential improvements for AspectJ is at the pointcut language, *i.e.,* the language used to define patterns of events to be matched against a main program's execution. In AspectJ, most pointcut definitions only consist of individual join points but not sequences of events. For instance, using AspectJ, it is impossible to define a pointcut matching three events e_1, e_2, e_3 precisely in that order. Although one can use aspect internal variables to capture the execution points where the sequence of events e_1, e_2, e_3 has occurred, being able to explicitly specify relationships between join points in pointcut declarations would be more convenient and more importantly, provide more information about the aspect for analysis purpose. Note that, AspectJ does support two control-flow-based pointcut designators: *cflow* and *cflowbelow* that are used to match join points in a specific control-flow. However, they are sufficient in general because many sequences to be matched do not necessarily include join points in the control-flow of another join point.

The second potential improvement for AspectJ is to provide support for property analysis. Since aspects are supposed to modify the execution of a main program, it is very important to be able to analyze the effects of aspects on a base program and verify the property of the integrated program to ensure correctness. The AspectJ's pointcut language is Turing-complete, *i.e.,* it allows pointcut definitions from arbitrary combination of join points. As a consequence, reasoning about pointcuts and aspects is very difficult or impractical. If the pointcut language is based on a less expressive language, for instance, regular language, analysis on pointcuts and aspects would be more feasible because analysis techniques for regular languages have already been well-developed. We believe that there is a compromise between the expressivity and the ability to support property analysis of a pointcut language. We aim at defining the pointcut language as expressive as possible but still keep it simple enough for the analysis to be feasible.

The third improvement we have considered is the support of the aspect language for component-based systems which are a promising application domain for AOP. AspectJ lacks dedicated support for component-based systems because it has been designed for general purpose only. AspectJ allows invasive modifications to base programs and thus might be inappropriate when applied over components in many cases. While protocols are widely used in component-based software to define constraints or contracts between components, AspectJ does not provide any specific designators for expressing component protocols. We believe that an aspect language that supports the definition of aspects over components should meet at least two requirements. First, pointcuts should be able to capture information available at the interface of a component, including method signatures and interaction protocols. Second, the effects of an aspect over a component should be transparent so that reasoning about the

integrated system can be done.

3.2.2 History-based aspect languages

This section covers one specific group of AO languages: history-based aspect languages. In general, all the languages among this group support pointcuts that can qualify join points based on what has happened before them. Aspects defined by history-based aspect languages can evolve through more than one state so they have been introduced as stateful aspects. In this thesis we will only use the term 'history-based'. These languages provide more sophisticate support to explicitly specify the temporal relationship between join points in a pointcut. The main difference between approaches in this group lies in the expressivity of their pointcut languages. They feature expressive but not necessarily Turing-complete pointcut languages so that property analysis over aspects can be supported. Pointcut languages of these approaches are usually backed by some kinds of state machines. For example, there have been some regular-based aspect languages [17, 48, 56], and very few non-regular (but not Turing-complete) ones [128].

In the following we discuss several important approaches in the group of history-based aspect languages. The section is divided into two main parts. Section 3.2.2.1 focuses on the generic framework for history-based aspects introduced by Douence et. al in 2002 [47] and further studies developed from this framework. Section 3.2.2.2 covers other independent approaches.

3.2.2.1 A generic framework for history-based aspects

Douence et al. [47] have introduced a generic framework for AOP which gives a foundation for the development of AO languages that support history-based pointcuts and the analysis for interaction between aspects. This framework is also the foundation for the aspect language we have developed.

An aspect language supported by this framework provides three basic constructs to define crosscutting concerns (*i.e.,* aspects): *crosscut*, *insert*, and *aspect*. A *crosscut* defines the collection of join points to be matched against a program's execution. An *insert* defines the code to be executed when the crosscut successfully matches the program's execution. Finally, an *aspect* is defined by the combination of basic rules of the form $C \rhd I$ where C is a crosscut and I is an insert. The most important characteristic of the aspect language defined by this framework is that it allows us to define crosscuts not only as sets of individual join points but also as sequences of join points. In other words, we can take into account previous join points that have occurred when qualifying a join point at runtime.

Let us return to the context of the data system example in the previous section. Assume that we would like to run an aspect that will run a virus check on all `commit` executions that follow an execution of the `add` method. This aspect is defined by the basic framework as follows:

$$\mu a_1.add \rhd skip \ ; \ commit \rhd check \ ; \ a_1 \ \Box \ terminate \rhd clearCache \ ; \ a_1$$

According to the above definition, the aspect starts by trying to match a join point of an `add` execution. When it matches an `add` join point, nothing is done (indicated by the `skip` instruction. The aspect then evolves to the next state where it tries to match a `commit` or a

`terminate` join point. If it matches a `commit` join point, it runs the `check` insert then evolves to the next state where it waits to match an `add` join point again (indicated by the end of sequence a_1). Otherwise, if it matches a `terminate` join point, it runs the `clearCache` insert and then evolves to the next state where it waits for another `add` join point.

Note that the $\mu a.A$ construct is the recursive definition of an aspect that is equivalent to A where all occurrences of a are replaced by $\mu a.A$. This definition is a form of tail recursion and thus the crosscut language can always be regular. The $\square$ symbol represents a choice between two aspects where the left operand is given preference in case both aspects match the same join point. Obviously, the ';' symbol indicates the temporal relation of join points.

Formally, the expressivity of the crosscut language defined by this framework is that of regular languages in the Chomsky hierarchy. Hence, it is expressive yet not Turing-complete. Since analysis techniques for regular languages are well-founded, the aspect language defined by this framework is more amenable to property analysis than aspect languages featuring Turing-complete pointcuts. A method for analyzing aspect interaction has been proposed and later elaborated for the generic framework [48]. This work will be discussed later in section 3.3.2.

In summary, the generic framework offers an interesting foundation for the development of stateful aspect languages, the detection and resolution for aspect interactions. There are, however, a number of potential advancements for the framework. First, the crosscut language in this framework only limits to regular languages. The most advantage of a regular-based crosscut language is that property analysis for aspects is practical. On the other hand, the drawback of a regular-based crosscut language lies in its limited expressivity. There are many complex control-flows, for instance, properly nested structures, that cannot be expressed using regular languages. Second, although the framework supports the declaration of protocols, which is useful for defining aspects over components, it lacks a collection of useful constructors for component protocols. Furthermore, an analysis technique for important compositional properties of components modified by aspects is not yet considered.

3.2.2.2 Other approaches

In this section, we present another two approaches for history-based aspect languages. Both introduce new constructs, *tracematches* in the first approach, and *tracecuts* in the second approach, as extensions to AspectJ to enable the declaration of history-based pointcuts.

Tracematches. Allan et. al [17] have introduced *tracematches*, a history-based language feature, as an extension to AspectJ to allow programmers to write code that can be executed based on a regular pattern of events in an execution of a program.

A tracematch is defined in a format very similar to an aspect in AspectJ. It includes a pattern and a code block to be executed when the current trace matches that pattern. Patterns declared in tracematches are described as regular expressions. Here is a tracematch that executes method `check` if it matches a trace that ends with a call to `add` or `remove` and a call to `commit`:

```
tracematch(){
    sym a after: call(* add(..)) || call(* remove(..));
    sym b after: call(* commit(..));

    a b {
```

```
6           check();
        }
8   }
```

We declare symbol a to match any `add` or `remove` events and symbol b to match any `commit` events (line 2-3). Line 5 shows the regular expression that specifies the pattern to match against the program trace. Line 6-7 shows the code black that will be executed if a match occurs. Note that only events that are declared as symbols can trigger the match. In other words, events that are of interest must be explicitly declared as symbols. For example, the pattern on line 5 matches the '`add commit`' sequence as well as the '`add update commit`' sequence since the `update` event is simply ignored from the matching process. Besides, it is possible to declare free variable in a tracematch and to have it bound to values captured by pointcuts.

Since matching patterns are described as regular expressions, the matching process for a tracematch is basically done by running a finite state automaton whose transitions are labelled by declared symbols in the tracematch. When a final state is reached, the advice code is executed. The implementation for the tracematches feature has been realized as an extension of the `abc` [1], the AspectBench compiler for AspectJ. Further details on the formal semantics of tracematches and the implementation can be found in the paper [17].

An advantage of this approach is that tracematches are amenable to static analysis as their declared patterns are regular expressions and the language inclusion problem for regular languages is decidable. However, there have been no work on this subject has been put forward. Similar to the previous generic framework for stateful aspects, the downside of tracematches is that regular expressions are not sufficient to capture many complex sequences, notably sequences with properly nested structures.

Tracecuts. The term *tracecut* has been introduced by Walker and Viggers in [128]. Their work aims at the definition and implementation of protocols via declarative event patterns (DEPs) which are specified as tracecuts in their language. Their language allows programmers to explicitly specify a protocol at a high-level and actions to be taken whenever the program trace matches that protocol.

The work on tracecuts was builded upon AspectJ. Two new constructs have been added to define protocol-based pointcuts: *tracecuts* and a *history* primitive pointcut. Tracecuts are specifications of declarative event patterns. Concretely, they are composed from entry and exit points of events captured by an AspectJ pointcut. Tracecuts are defined using context-free grammar and thus can express regular protocols as well as context-free protocols, especially protocols involved nested structures. The *history* primitive pointcut receives a tracecut as an argument. When it is used in a pointcut of an advice, the action defined in the advice body is executed if the actual execution matches the pattern described in the tracecut argument of the history pointcut.

Let us consider a small example (which is a slightly modified version of the example in [128]) of a protocol that involves three methods namely `safe`, `unsafe`, and `commit`. The `safe` and `unsafe` methods can be called in a mutually recursive structure, *i.e.,* `safe` can make a call to `unsafe` and vice versa. Both `safe` and `unsafe` methods can invoke `commit` to do something which then would be advised differently by an aspect depending on whether it is called in a safe or unsafe context. The following code represents the above setting:

```
pointcut safePC(): execution(* safe(..));
```

```
2   pointcut unsafePC(): execution(* unsafe());
    pointcut commitPC(): execution(* commit());
4
    tracecut completed() ::=  (entry(safePC()) [completed()] exit(safePC())) |
6                     (entry(unsafePC()) [completed()] exit(unsafePC()));

8   tracecut isSafe() ::= entry(safePC()) complete()* $;

10  void around(): commitPC() && history(isSafe()){
        System.out.println("Commit in safe context");
12  }
```

In the above code, line 1-3 define three primitive pointcuts capturing excutions of three method `safe`, `unsafe`, and `commit`. The `completed` tracecut (line 5) captures all completed and properly nested pairs of `safe` and `unsafe`. Since we need to recognize `commit` events triggered by `safe`, we define the `isSafe` tracecut to capture the context that is more tightly enclosed in an execution of `safe` than `unsafe`. The '* $' part matches the end of the execution trace up to the current join point. Hence, the `isSafe` tracecut matches the execution trace only if there is un unmatched entry to `safe` and no unmatched entries to `unsafe` occurs before the current join point. Line 10-13 indicate that if the `commit` method is executed when it is in a safe context captured by `isSafe`, the (dummy) advice code can be run.

A proof-of-concept tool has been realized to translate aspects with new constructs for DEPs into standard AspectJ pointcuts and advice. Basically, since a tracecut is described by a context-free grammar, matching a *history* primitive pointcut relies on running a pushdown automaton representing its tracecut argument along with the execution to recognize events. Those interest in the details of the implementation can refer to [127].

This approach provides a means to directly express protocols in an aspect pointcut. Most importantly, it allows for the specification of protocols involved properly nested structures that no previous history-based aspect languages have offered. However, from a point of view of program analysis, this feature has a disadvantage that it impedes analysis support since the inclusion problem is undecidable for context-free languages.

3.2.3 Adaptive programming

The Adaptive Programming approach [83] has been built upon the key principle defined by the Law of Demeter [80]. That law states in its general form as "Each unit should have only limited knowledge about other units. Only units closely related to the current unit should be talked to" [5]. In short, this Demeter rule makes explicit one special case of the loose coupling principle for the object-oriented programming paradigm: objects should not depend on the internal structure of other objects. For example, an object A can request a service offered by another object B but A should not make a request for services provided by another object C which can only be reached through B as that would mean A has to depend on the internal structure of B.

Adaptive Programming provides mechanisms to control the coupling between structural and behavior concerns of a program. Basically, an adaptive program is written in terms of loosely coupled contexts where data structures and computations (or behaviors) are two contexts which are bound by a third definition. Adaptive methods are introduced to encapsulate the behaviors of operations in one place and also abstract over the class structures on which

they apply. Therefore, adaptive methods not only help avoiding the scattering problem but also the tangling problem [83].

An adaptive method essentially consists of two parts: a traversal strategy and an adaptive visitor. A traversal strategy defines paths on the program data structure to which visitors can be attached. An adaptive visitor implements the computations that are executed at visited classes. The following sample code, taken from [83], illustrates the structure of a simple adaptive method implemented using the DJ library [101], a Java library that supports writing adaptive methods.

```
import edu.neu.ccs.demeter.dj.ClassGraph;
import edu.neu.ccs.demeter.dj.Visitor;

class Company{

    // class structure
    static ClassGraph cg = new ClassGraph();

    Double sumSalaries(){

        // traversal strategy
        String s = "from Company to Salary";

        // adaptive visitor
        Visitor v = new Visitor() {
            private double sum;
            public void start() {sum = 0.0};
            public void before(Salary host) {sum += host.getValue();}
            public Object getReturnValue() { return new Double(sum);}
        };

        return (Double) cg.traverse(this,s,v);
    }

    // rest of Company definition
}
```

The above code shows the adaptive method namely *sumSalaries* (line 9-23) in class *Company*. This method counts the total salary of all employees in a company. The string *s* (line 12) represents the traversal strategy applying over the class graph *cg* which represents the class structure of the company. That is, the adaptive visitor will visit numerous locations in the class structure using that traversal strategy. The adaptive visitor (line 15-20) defines one *before* advice (line 18) that updates the total salary of all employees by adding the salary at every visited location in the class structure. Since the traversal strategy allows for the abstraction of how the visitor traverses the class graph starting from the *Company* object to the *Salary* object, the adaptive method can be reused over many different company structures, no matter how many departments or affiliates or employees a company has.

A few implementations have been created for the Adaptive Programming approach. DemeterJ [82] adds special language constructs to Java to support adaptive programming. DJ [101] is a Java library that allows programmers to write adaptive methods. DAJ [119, 81] is the most recent tool to offer special constructs to support adaptive programming to AspectJ. Traversal strategies in DAJ are regular expressions.

Modifications to program data structure may or may not alter the overall behavior of the problem. In order to cope with the problem in which modifications affects the meaning of the program, Demeter Interfaces [117] are introduced to adaptive programming. Demeter Interfaces make explicit the required structural properties of the data structure in order for adaptive methods to function properly.

3.2.4 Composition filters

Composition Filters model [13, 14], originating from the Sina language [15], is one of the earliest AOP approaches. This approach relies on the use of filters to improve the current existing object-oriented programming languages in two aspects: composability, *i.e.,* the ability to compose existing modules into new modules, and evolvability *i.e.,* the ability to modularly extend existing program modules. In the following we present the principle concepts and ideas of the Composition Filters model, and give an example to illustrate how filters can be defined and used in this model.

In this approach, a set of filters are introduced to apply over the primary behaviors of a system in order to adapt or extend the original behaviors. More precisely, conventional objects (implemented using traditional object-oriented languages) are enhanced by filters manipulating sent and received messages to and from these objects. Each filter specifies the condition to match the messages that are of concern and the manipulation that will be applied to the matched messages.

Let us consider an e-commerce application. In this application, a customer can place orders and have his orders delivered to his address after payment. At the implementation level for this application, all the operations related to orders are implemented in class *Order*. For simplicity, lets assume that class *Order* consists of four methods: *placeOrder* (to put place a new order), *cancel* (to cancel an existing order), *deliver* (to deliver items to customers), and *setPaidStatus* (to update the paid status indicating whether an order has already been paid or not). Then these four methods are invoked by instances of class *Customer* (representing customers), class *Accounting* (representing the Accounting department that verifies payment details and updates paid status for orders), and class *Delivery* (representing the Delivery department that is responsible for shipping items to customers). Obviously, a *Customer* object should be able to invoke *placeOrder* and *cancel* methods only, and similarly for *Accounting* object with *setPaidStatus* method, and *Delivery* object with *deliver* method. The Composition Filters approach realises these restrictions by extending class *Order* with a layer that contains a set of filters for manipulating invocations to its methods. The following code gives an example of the filter module for handling the concern of restricting access to class *Order*.

```
concern ProtectedOrder begin
filtermodule OrderWithViews begin
    internals
        order: Order;
    conditions
        Customer; Accounting; Delivery;
    methods
        order.*;
    inputfilters
        protection: Error = { Customer => {placeOrder, cancel},
                    Accounting => {setPaidStatus},
                    Delivery => {deliver}};
```

```
            ihn: Dispatch = (inner.*, order.*);
14      end filtermodule OrderWithViews;

16      implementation in Java
            class ProtectedOrderImpl{
18              boolean Customer() {...//true if the invoking object is of Customer class};
                boolean Accounting() {... };
20              boolean Delivery() {...};
            }
22      end implementation
        end concern ProtectedOrder;
```

The filter module *OrderWithViews* declares an instance of class *Order* as its internal object (line 3-4) since it is used to encapsulate objects of class *Order*. In other words, the filter module can manipulate incoming and outgoing messages made to and from an *Order* object. Three conditions namely *Customer*, *Accounting*, *Delivery* are declared (line 5-6) to abstract the state of the implementation object. That is, they indicate whether a method invocation (a received message) is made by an object of the *Customer* or *Accounting*, or *Delivery* class. These conditions are realised as boolean methods defined in the implementation part of the extended class (line 18-20). The third part of the filter module (line 7-8) specifies methods that are present in the filter expressions later on. Then two (input) filters namely *protection* and *ihn* are specified to handle the task of restricting access to an *Order* object (line 9-13).

A filter typically consists of a *filter type* and a *filter pattern*. The filter type expresses how the messages are handled if they successfully match the filter pattern. A number of different filter types can be defined to suit specific needs for message filtering. The filter pattern is a simple expression to match incoming and outgoing messages to and from objects. In the above code, two input filters (filters that manipulate incoming messages) are declared: *protection* filter and *ihn* filter. The *protection* filter is of type *Error*. That means, if the message is rejected, an exception is thrown; otherwise, the message is passed to the next filter. For example, filter *protection* specified that if the incoming message is sent by a *Customer* object, the message should match either *placeOrder* or *cancel*, which are the only two methods that a *Customer* object is allowed to use. Conversely, if the message is sent by a *Customer* object but it does not match any of these two methods, the filter will raise an exception. Hence, if a message is either accepted by the *protection* filter or not sent by one of the three object types, *Customer*, *Accounting*, and *Delivery*, it travels to the *ihn* filter. The *ihn* filter (line 13) is a *Dispatch* filter *i.e.,* if a message is accepted, it is dispatched to the target of the message, *i.e.,* the *Order* object; otherwise, it travels to the next filter. For instance, if a message is a method of class *Order* (specified as `order.*`), the filter accepts the message and dispatches the message to the corresponding *Order* object, *i.e.,* the method then can be excuted.

The above example shows how filters can be applied to the *Order* class. Besides, filter modules can be composed to address crosscuting concerns involving several classes through superimposition (see [14] for an example).

In comparison to AspectJ aspects, composition filters are more independent of the implementation of the classes on which they are applied. Their specifications are restricted to the interface level. Therefore they are easier to be reused than AspectJ aspects. Furthermore, they do not introduce invasive modifications to the internal behaviors of the target classes but only extensions to those classes through manipulations of incoming and outgoing messages. Composition filters also do not refer to each other. However, composition filters are less ex-

pressive than AspectJ aspects. Besides, languages and tools for composition filters approach are not as well developed and mature as AspectJ.

Since the Composition Filters model uses a restricted pattern matching language rather than a general-purpose programming language, it provides opportunities for reasoning about the properties of concerns specified as filters. For instance, a study on techniques to detect and correct interactions among aspects at the same join point has been reported in [50].

3.2.5 Summary

We have described the major AO approaches that have been developed since the SoC principle has become an accepted idea. Among all AO approaches, AspectJ remains the most mature and user-friendly language which is practical for building real-world applications. Approaches on history-based AO languages offer the ability to capture sequences in the declaration of patterns of crosscutting concerns. This characteristic helps the AO languages to have the potential for property analysis which AspectJ does not have. Interaction analysis for aspects has been one of such analysis that has been studied so far for the generic framework on stateful aspects. Finally, approaches on adaptive programming and composition filters contribute valuable ideas on establishing the interface between aspects and the main program to achieve loose-coupling principle and aspect reuse.

3.3 Verification and Analysis

One of the critical advantages of AOP is the ability to perform invasive modifications to a program's execution. However, this also makes it more difficult to predict the behavior of the composed program. Furthermore, aspects may even cause insidious errors by violating properties of the base program. As a consequence, a number of approaches on verification and analysis for aspect-oriented programs have been proposed to help reasoning about the properties of aspects and programs modified by aspects. These approaches usually focus on, but not limited to, the following issues: whether an aspect produces a desired property for the integrated program, whether a property of a base program is preserved after the application of an aspect, whether aspects interfere each other when both are applied to the same base program.

This section discusses two major groups of verification and analysis support for AOP. The first group consists of approaches where model checking techniques are employed to provide computer aided verification for aspect-oriented programs. The second group consists of approaches where static analysis techniques are developed on aspect language models in order to statically analyzed properties of aspect systems.

3.3.1 Aspect verification using Model checking

Model checking is a popular technique that enables automatic verification of finite-state systems. Generally, a model checker consumes a description of a system and a specification of a property that must be satisfied by the system and then performs the verification process in order to conclude whether or not the system satisfies the property. If the system does not satisfy the property, counter examples can be generated to indicate where the property is violated. An input system is usually described as a finite state machine and a property

is typically specified in a temporal logic such as LTL, CTL. At the moment, SPIN [65] and NuSMV [38] are among the most popular model checkers that are freely available.

The application of model checking has been very successful for hardware verification. It is, however, much more difficult to use model checking technique in software verification due to the complexity of a software. Note that model checking is based on state space exploration technique. If a software system is too complicated, the size of the state space to be explored may be enormous and thus exploration may be infeasible (this problem is called state space explosion in model checking). Therefore, it is very important to optimize the size of the system to be verified.

In recent years, there have been quite a number of research conducted in AOP community that consider using model checking for the verification of aspect-oriented programs, for examples [77, 73, 76, 45, 116, 59]. When model checking is applied to the verification of aspect-oriented programs, the most naive approach is to compose the base program and aspects to produce the complete input system for a model checker then model check that system. However, such an approach is not efficient for aspect programs since every time a new aspect is introduced to the system, the whole program has to be composed and the verification has to be done again from the beginning. Note that the verification of the whole system may take a lot of time if the system is complex. Furthermore, programmers who write and verify aspects have to be aware of the internals of the base program which could be a hard-to-fulfill requirement in many cases. As a consequence, studies on verification approaches for aspect-oriented programs usually aim at modular ways where aspects can be verified separately from the base program in order to improve such complicated situations.

In the following, we review two model checking approaches for aspect-oriented programs. The first one provides a modular LTL model checking method for aspect-oriented programs published by Goldman and Katz [59]. The second one is the work based on CTL model checking presented by Krishnamurthi and Fisler [76].

3.3.1.1 Modular LTL model checking

Goldman and Katz [59] present a framework for the generic formal verification of aspects relative to their specification. The authors have proposed to integrate the specification of an aspect with the assumptions about the types of base programs to which the aspect can be woven. The underlying purpose of such a specification is to be able to use those assumptions as a representation for a base program and to build the augmented system as a composition of such assumptions and the actual aspect in order to verify whether a desired property is satisfied by that augmented system. In other words, aspects can be verified separately from the actual base program and thus verification does not have to be done again in the case of changes to the base program as long as that program still satisfies the assumptions made by the aspect.

This verification approach is strongly based on LTL model checking technique. In principle, given a state machine that represents the input system, LTL model checking is done by creating another state machine that accepts exactly computations that satisfy the LTL property then calculating the cross-product of the negation of that state machine (which represents all the paths that violate the property) and the model of the input system to be checked. If the resulting state machine from the cross-product calculation is not empty, conclusion can be made that the property is violated by the system.

Concretely, the verification model of this framework involves the following three basic

components:

- **Base program:** This verification approach uses the idea of LTL model checking in a unique way: to build the state machine representing the class of base programs from assumptions specified as LTL formulas. Hence, the base program in the verification model is represented by the state machine constructed from the LTL formula which specifies assumptions of aspects over the base program.

- **Aspect:** Pointcuts of an aspect can be defined using regular expressions or LTL formulas over atomic propositions about the states of the program state machine. Advice of an aspect is also represented by a state machine. Note that, this approach only handles *weakly invasive* [72] aspects, *i.e.,* aspects that return to a state that was already reachable in the original base program.

- **Property:** Properties in this verification model are desired properties of the augmented system composed from the base program and the aspect and they are specified as LTL formulas over atomic propositions about the states in the program state machine.

The goal of the verification is to check whether the augmented system satisfies a specific LTL property. Before the actual verification may start, necessary state machines are constructed to provide inputs for the verification algorithm. First, from the formula ψ that expresses the assumptions about the base program, a state machine T_ψ that includes all paths that satisfy ψ is created. Next, aspect A is woven into the just created state machine T_ψ according to pointcut descriptor ρ. This weaving process basically requires a transformation of the state machine T_ψ to a pointcut-ready state machine so that it is possible to separate paths which match ρ and those that do not. Then the advice state machine is integrated to that pointcut-ready state machine to create the augmented model $\widetilde{T_\psi}$. Finally model checking is performed on model $\widetilde{T_\psi}$ and property ϕ to determine whether the model satisfies the property, *i.e.,* $\widetilde{T_\psi} \models \phi$.

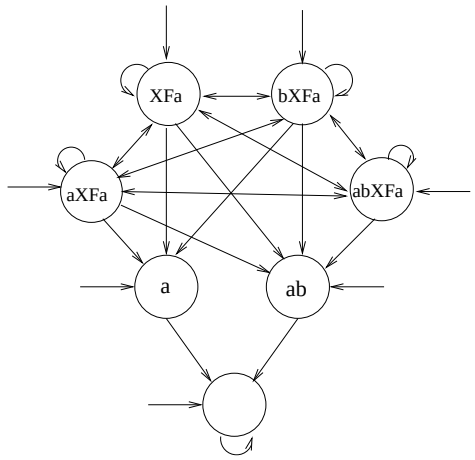
Let us consider the following example (extracted from [59]) for a demonstration of the above verification process. The values for assumption formula ψ and property ϕ over the set of atomic propositions $AP = \{a, b\}$ are as follows:

$$\psi = AG((\neg a \wedge b) \longrightarrow F\ a)$$
$$\varphi = AG((a \wedge b) \longrightarrow X\ F\ b)$$

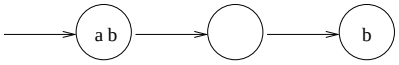
Basically, the assumption ψ can be interpreted as that for all possible paths in the base program any state satisfying $\neg a \wedge b$ will be eventually followed by a state satisfying a. The state machine T_ψ constructed from ψ is presented in figure 3.1(a).

The property ϕ to be verified is that for all possible paths in the augmented system, any state satisfying $a \wedge b$ will immediately followed by a state satisfying a. Assume that the pointcut descriptor ρ for our aspect is $a \wedge b$ and the aspect advice is represented by the state machine A in figure 3.1(b). Weaving is done by integrating the advice state machine into program state machine T_ψ at the states that satisfy pointcut descriptor ρ to create the augmented state machine $\widetilde{T_\psi}$ shown in figure 3.3.1.1. Apply model checking to the resulting state machine $\widetilde{T_\psi}$ with property ϕ would prove that $\widetilde{T_\psi} \models \phi$.

This verification approach has been implemented in a prototype tool namely MAVEN. This tool takes a specification of the assumption, the property and the aspect as textual inputs and



(a) State machine T_ψ constructed from $\psi = AG((\neg a \wedge b) \longrightarrow F\ a)$



(b) State machine representing aspect advice A

Figure 3.1: State machines representing base program requirements and aspect advice

then uses an LTL model checker to generate the state machine from the assumption. Next, the tool weaves aspect advice into the program state machine and feeds the augmented state machine and the LTL property into existing model checker NuSMV [38] for the verification.

The strength of this approach lies in its ability to verify an aspect for a class of base programs rather than one specific base program. This means that the approach achieves modular verification at a certain level: verification on aspect can be done independently as long as the assumptions about the base program is written in the specification of the aspect. However, at the moment, the process to weave the aspect to the base state machine is very complicated and time consuming when the base program and the pointcut are not trivial. Moreover, the prototype implementation shows that this process can just be done for very simple pointcuts.

3.3.1.2 Incremental CTL model checking

Krishnamurthi and Fisler [76] have presented an interesting theoretical framework for applying model checking to aspect-oriented programs. Their approach aims at identifying cases where aspects may violate the desirable properties of the base program. Their analysis technique is modular in the sense that verification has to be done again only when certain critical parts of the program change and unnecessary repeated analysis is avoided in other cases.

The approach heavily depends on CTL model checking technique [39]. CTL (Computation Tree Logic) is a branching time logic which allows for the expression of possible futures in a tree-like structure. CTL formulas can be defined using standard atomic propositional operators, plus temporal connectives which are used to describe possible future paths from the current state. CTL model checking relies on the labeling algorithm which, given a model and a CTL formula, calculates the set of states that satisfy the formula. During the run of the labeling algorithm, the CTL formula is traversed bottom-up while each state of the model is labeled with all sub-formulas that are true at that state. By doing that, when the labeling algorithm terminates, all states are labeled with all sub-formulas of the CTL formula and thus we can obtain the set of states that satisfy the property.

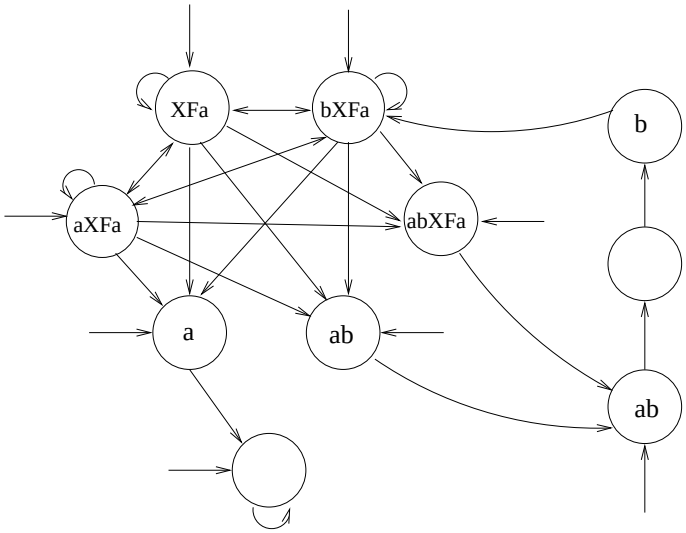


Figure 3.2: State machine $\widetilde{T_\psi}$ composed by weaving A into T_ψ according to $\rho = a \wedge b$

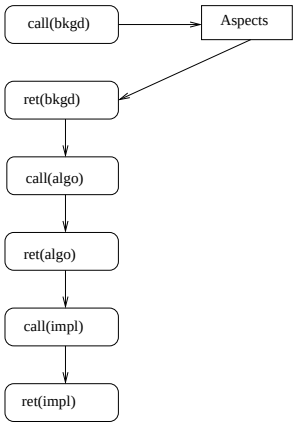
Similar to most other approaches on aspect verification, the verification model for this approach contains three basic components:

- **Base program:** Informally, a base program is represented by a state machine of which states associate to program statements or expressions while transitions associate to the control flow between statements or expressions in the base program. Besides, there are extra states to reflect the entry and exit points of program fragments respectively.

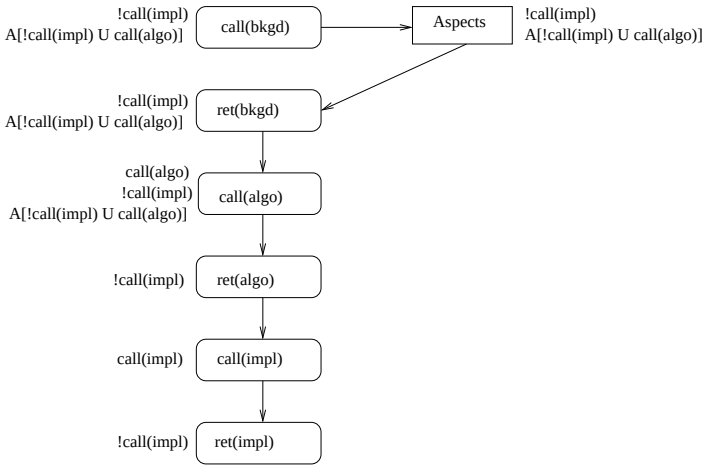
- **Aspect:** Current work only consider join points which correspond to function calls. In such context, pointcuts are defined by a subset of regular expressions over function calls. Pointcuts in this framework describe the shape of the stack at program states rather than sequences of calls leading to states. Hence, states where the current stack could match a pointcut are locations to which advice may apply. Advice is represented by a state machine with additional states to represent proceed and resume points (similar to entry and exit points in program fragments).

- **Property:** Properties in this framework are invariants of the base program and we would like to verify whether or not those properties are preserved after the application of aspects. Properties are specified as CTL formulas.

To have a more intuitive presentation of the above verification model, consider a simple base program implementing a slideshow presentation [76]. This program consists in a sequence of three function calls to *bkgd*(background), *algo*(algorithm), and *impl*(implementation) in this order. Normally, contents on algorithm and implementation are common for all of the talks so they can be implemented statically in the base program. On the contrary, the background content may vary greatly depending on the target audiences. Therefore, it could be implemented in a more cleanly way by using aspects to insert appropriate contents when needed, *i.e.,* different aspects are provided to present different background contents and they are triggered after there is a call to *bkgd* function in the base program. Figure 3.3(a) illustrates the state machine of the slideshow program as described above.



(a) State machine of the slideshow program



(b) Program annotated with CTL labels

Assume that the property of the program we would like to preserve in all cases is that the section on implementation should always come after the algorithm description. This property is defined by the following CTL formula:

$$A[!call(impl)\ U\ call(algo)]$$

The whole verification process starts by checking the desired property on the base program (since the ultimate goal of the verification is to verify whether aspects preserve this same property of the base program). After this phase, each state of the base program state machine is labeled with subformulas of the property that are true at that state. Figure 3.3(b) shows the slideshow program state machine annotated with labels which are subformulas of the property on each state. The labels on the states where advice may apply are then reused to provide information about the state of the base program for later verification on the aspects. In our example, labels on states related to call and return to *bkgd* are reused since they are states that lead to and return from the advice.

The verification algorithm then checks whether the labels on the state before advice application hold on the *in* state of the advice while assuming that the labels on the state after advice application hold on the *out* state. In principle, this verification algorithm verifies the advice state machine without traversing the augmented state machine since it only considers certain parts of the base program's state machine where advice may apply. However, handling cases of around advice or advice triggered by other advice is more complicated.

The major limitation of both model checking approaches, especially the second one, is that they are mainly theoretical results and only applicable for very simple applications. Although there is a tool that has been implemented for the first model checking approach, that tool is still just a proof-of-concept prototype that only realizes the simple part while skipping the complicated one of the checking algorithm.

3.3.2 Static analysis techniques for aspects

In this section, we discuss a variety of static analysis approaches that have been proposed for aspect-oriented programs. In these approaches, properties of aspects and integrated pro-

gram are established using dataflow analysis techniques and identifying aspects to determine whether they belong to classes for which some properties can be automatically guaranteed.

3.3.2.1 Classification of aspects

Properties of the augmented program from the base program and aspects may change differently depending on the types of interactions between aspects and base programs. There have been proposals for classifying the relation between an aspect and a base program and proving properties for certain classes of relations. Such classification systems enable reasoning about aspects and understanding the behaviors of the integrated program. If we can classify an aspect into a certain class for which some properties have been proved then we may conclude a property for the augmented program. The advantage of this approach is that it may reduce the cost of reasoning about certain classified aspects. However, it may not possible to apply this approach for all kinds of aspects. Therefore, the goal is to define restricted by still general classes of aspects in order to include as many as possible aspects that can fit into the classes for which some properties can be established.

The first classification of aspects proposed by Sihman and Katz [115, 72] divides aspects into three types based on how they affect the base program: *spectative*, *regulative*, and *invasive* aspects. Spectative aspects only gather information about the base program and do not change the fields or the control-flow of a base program. Regulative aspects can change the control-flow but not the existing fields of the base program. Finally, invasive aspects can change both the control as well as the existing fields of the base program. Certain properties have been proved for three above classes [72].

Rinard et al. [109] have proposed a similar but more refined categorization that classifies the interactions between an aspect and a method of the base program into five classes: orthogonal (both may access disjoint fields), independent (neither may write to a field that the other writes), observation (the advice may read fields written by the method), actuation (the method may read fields written by advice), and interference (both write to the same fields). The authors have also put forward an analysis tool to automatically recognize these interaction classes.

3.3.2.2 Analysis for aspect interaction

Beside approaches concentrating on the relation between an aspect and a base program, there have been a body of works, *e.g.,* [47, 48, 60], on detecting interactions among aspects that are applied to a same base program. In the following we present the method that has been proposed for the generic framework described in section 3.2.2.1. Since we have developed our aspect language based on that generic framework, we could also apply and extend the analysis technique for detecting aspect interactions proposed for it.

The goal of interaction analysis for aspects is to detect when the parallel composition of aspects does not guarantee a deterministic weaving. This happens when two aspects match the same join point. If two aspects are independent they do not interact and can be woven in any order. Otherwise, if two aspects do interact and the order of weaving may lead to different results then a resolution is needed to settle the conflict. For instance, assume that we define two aspects that implement two separate concerns as follows:

$$A_1 = \mu a.commit() \rhd acquireConnection() \ ; \ a$$
$$A_2 = \mu a.add() \rhd skip \ ; \ commit() \rhd checkVirus() \ ; \ a$$

These above aspects will match the same `commit` join point at runtime so we say that they interact by definition. Whether a conflict resolution is needed depends on whether the order of weaving, *i.e.,* execution of inserts, produces different results or not. For example, if the `acquireConnection` is a resource-consuming service we might wish to weave A_2 before A_1 so that if the virus check does not pass, we do not need to continue with the other insert.

Two notions of independence have been introduced [47]: strong independence and weak independence. Strong independence does not depend on the program to be woven. Therefore, analysis for strong independence among aspects can be done without any information on the actual base program and thus the result is not affected by program modifications. In contrast, weak independence depends on the program to be woven. This property is obviously a less strict condition to enforce but analysis for weak independence is more costly since the actual base program that the aspects will be woven into has to be considered in the analysis.

The analysis for strong independence between two aspects is performed based on a set of composition laws [48]. Basically, two aspects will be unfolded so that we may obtain a picture of how two aspects work in parallel. This unfolding technique is similar to a calculation of the product of two finite state automata. Next, traces that end in similar join points triggering simultaneous application of inserts are marked as potential interactions (or conflicts) between two aspects. Note that such check for strong independence can be done thanks to the key characteristic of the crosscut language: regular-based, as it has been known that we cannot do the similar product calculation for more expressive languages which are non-regular.

Two types of conflict resolutions have been put forward: sequential composition operator and composition adapter for aspects. The sequential composition operator is defined as $A_1 - C \rightarrow A_2$ which indicates that aspect A_1 is applied until an event is matched by crosscut C then A_1 is stopped and A_2 is started. Consider the following two aspects A_1, A_2 which would interact at all `commit` join points if composed in parallel:

$$A_1 = \mu a.commit() \rhd checkAll() \ ; \ a$$
$$A_2 = \mu a.commit() \rhd checkVirus() \ ; \ a$$

where A_1 will run a complete check for correct authorization and virus risks while A_2 will only run a check for virus risks. We can refine the composed aspect using the sequence operator $A_1 - secureConnect() \rightarrow A_2$ so that a complete check is performed for every `commit` execution until the user acquires a secured connection then only a virus check is needed for a `commit` execution.

The composition adaptor is defined to guide the composition to yield deterministic aspect weaving. Consider the following aspects A_1, A_2 which would also interact at all `commit` join points:

$$A_1 = \mu a.commit() \rhd checkAll() \ ; \ a$$
$$A_2 = \mu a.login() \rhd secureConnect() \ ; \ \mu b.commit() \rhd checkVirus \ ; \ b$$
$$\Box logout() \rhd closeConnection() \ ; \ a$$

where A_1 runs complete checks for all `commit` executions and A_2 creates a secured connection after a user logs in and only runs virus checks for all `commit` executions commanded by this user. The parallel composition yields conflicts at `commit` join points whenever a `commit` execution occurs in a secured connection. The following composition adaptor O is defined to resolve this conflict:

$$O = \mu a.login() \rhd (id, skip) \ ; \ \mu b.commit() \rhd (id, snd) \ ; \ b \\ \Box logout() \rhd (id, skip) \ ; \ a$$

The above definition indicates that if only one aspect matches a `commit` join point the insert defined for that aspect is executed (indicated by term *id*). If two aspects match the same `commit` join point the insert of the second aspect is chosen (indicated by term *snd*). The composition which O is used as an adapter yields the following aspect:

$$A_{12} = \mu a.commit() \rhd checkAll() \ ; \ a \\ \Box login() \rhd secureConnect() \ ; \ \mu b.commit() \rhd checkVirus() \ ; \ b \\ \Box logout() \rhd closeConnection() \ ; \ a$$

which would run `checkAll` at all `commit` states until after a *login* join point is matched.

In addition to interaction analysis among aspects, the regular-based crosscut language also enables the analysis between an aspect and a base program. For instance, given an abstract regular control-flow of a base program, one can verify where an aspect would be applicable to that base program by simulating the composition of the base program's abstract control flow and the aspect's crosscut. Consequently, we can specify the applicability condition of an aspect for a base program as a sequence of join points. As long as a concrete base program satisfies the applicability condition then we can conclude that the aspect is applicable for the base program. This enables aspect reuse. Furthermore, taking into account the information about the control-flow, either abstract or concrete, may help to prove independence for two aspects that only interact on execution traces which never occur when they are applied to a concrete base program.

3.3.3 Summary

We have presented a number of verification and analysis approaches that have been developed for aspect-oriented programs. These approaches have established a foundation for the development of methods for analyzing aspects and augmented programs. However, since most of them were developed for simple and generic aspect languages, they need to be adapted and extended in order to be applicable to more advanced or specific aspect languages and application domains. Note that verification on the augmented program is usually expensive or even impractical because of the complexity of the augmented system. Therefore, approaches on verification and analysis should enable modular reasoning in which aspects and base programs can be verified separately. Most of the above approaches aim at achieving this requirement. However, their results are still basically theoretical results and just applicable on trivial examples.

3.4 Aspects and Components

In this section we present a few major AO approaches for the implementation of aspects over components, including JBoss/AOP [69], SpringAOP [12], JAsCo [120], and CaesarJ [91, 23]. We will briefly describe key constructs and features for each approaches along with some examples to demonstrate those features.

3.4.1 JBoss/AOP

JBoss/AOP [69] is an extension of JBoss [7], an open-source J2EE application server, to support AOP paradigm. Although JBoss/AOP has been created to be used in the JBoss application server, it is an independent framework and thus can also be used in any Java program. In the following we briefly present how aspects are written in the JBoss/AOP framework.

In JBoss/AOP, aspects are implemented as *interceptors* that are capable of intercepting invocations to component services. An interceptor is simply a Java class implementing the *org.jboss.aop.Interceptor* interface. Hence, an interceptor has to define two methods as required by the *Interceptor* interface: *getName()*, which is used for display and *invoke(Invocation)*, which is where the advice code is put. The argument of method *invoke* carries the actual invocation that would have occured during the execution of the base program. The following code fragment illustrates the definition of an JBoss/AOP aspect namely *LoggingInterceptor* that simply prints the name of a method invocation for logging purpose.

```
1   public class LoggingInterceptor implements Interceptor{
        public String getName() {
3           return getClass().getName();
        }
5
        public Object invoke(Invocation invocation) throws Throwable {
7           if(invocation.getType() == InvocationType.METHOD){
                MethodInvocation mi = (MethodInvocation)invocation;
9               System.out.println(mi.method.getName());
            }
11
            return invocation.invokeNext();
13      }
    }
```

JBoss/AOP interceptors can be bound to pointcuts through XML configuration files. The following code shows a configuration file where the LoggingInterceptor is attached to all methods in all classes in the `acc` package.

```
    <aop>
2       <bind pointcut="* acc.*->*(..)">
            <interceptor class="LoggingInterceptor"/>
4       </bind>
    </aop>
```

JBoss/AOP provides a rich set of pointcut expressions that can be used to define many complex pointcuts. Like many other AOP frameworks, JBoss/AOP aspects can introduce invasive modifications to components. This may cause incompatibility between components if

such modifications are not well documented and properly checked. Besides, it does not support the explicit declaration of history-based pointcuts or pointcuts over component protocols.

3.4.2 Spring AOP

Spring AOP is one of the key components of the Spring Framework [12], a platform for the development and execution of enterprise Java applications. AOP can be used in the Spring framework to implement aspects that are responsible for enterprise services, in particular, transaction management services as well as custom aspects written by users [53].

Spring AOP only supports method interceptions. In other words, join points in Spring AOP always represent method executions on Spring beans (Spring components). In Spring AOP, aspects can be written using one of the two approaches: (1) the schema-based approach, or (2) the @AspectJ style. In the schema-based approach, aspects are declared using the Spring XML configuration style. The AspectJ style refers to the declaration of aspects as regular Java classes with Java 5 annotations. In the following we present an example of aspect implemented in Spring AOP using AspectJ annotation style.

```
1   package org.xyz;
    import org.aspectj.lang.annotation.Aspect;
3   import org.aspectj.lang.annotation.After;

5   @Aspect
    public class Logging{
7
        @Pointcut("execution(* acc.*.*(..))")
9       public void anyAccServices() {}

11      @AfterReturning("anyAccServices()")
        public void writeLog(){
13          // write to log file
        }
15  }
```

The above aspect namely *Logging* simply writes logging data to files after every execution of any method of any class in the *acc* package. The pointcut namely *anyAccServices* is declared by a pointcut expression (line 8) which matches the aforementioned condition and a pointcut signature (line 9). Since the pointcut is declared as public, it can also be refered (and reused) outside of the aspect *Logging*. The advice namely *writeLog* (line 11-14) is an *after-returning* advice. It associates with pointcut *anyAccServices()* (line 11) and executes its body after the execution of any method matched by its pointcut completes normally.

Similar to the Jboss/AOP framework, Spring AOP does not provide any mechanisms to implement protocol-based aspects. The pointcut language is that of AspectJ so it is more expressive while support for property analysis on aspects and components is more limited in comparison to history-based aspect languages discussed in section 3.2.2.

3.4.3 JAsCo

JAsCo [120] is an aspect-oriented language designed especially for component-based software development. The JAsCo language, while introduces a few new keywords and constructs for the implementation of aspects, is kept as close to the regular Java language syntax as possible.

JAsCo has been developed based on the features of two existing AO technologies: AspectJ [74] (which has been previously presented in section 3.2.1) and Aspectual Components [79].

The JAsCo language introduces two concepts: *aspect beans* and *connectors*. Aspect beans implement concerns that normally crosscut several components in the system. The following code [120] shows an JAsCo aspect bean that performs access-control tasks.

```
1   class AccessManager{
        PermissionDb pdb = new PermissionDb();
3       User currentuser = null;

5       hook AccessControl{
            String exceptionmessage = "General Access Exception";
7
            AccessControl(method(..args)){
9               execution(method);
            }
11
            around(){
13              if(pdb.check(currentuser,thisJoinPointObject))
                    return proceed();
15              else
                    throw new AccessException(exceptionmessage);
17          }

19          void setExceptionMessage(String aMessage){
                exceptionmessage = aMessage;
21          }
        }
23  }
```

The above aspect bean is declared as a regular Java bean that declares one hook namely *AccessControl* (line 5-26) as a kind of inner class. The hook consists of two parts: (at least) one constructor (line 8-10) specifying the condition when the normal execution of a method should be advised and one behavior method namely *around* (line 16-21) implementing the behavior to be executed (advice). The hook constructor is described by its abstract method parameters as input arguments (line 8) and its body (line 9). In this case, the hook constructor specifies that the behavior method(s) of the *AccessControl* hook will be performed whenever a method taken as an input of the hook is executed. Here, the behavior of the hook is implemented by the *around* method (line 16-21) which checks whether the current user is permitted to perform the method being executed. The *around* method would raise an *AccessException* if the check fails. Otherwise, the advised method can proceed.

The above JAsCo model with aspect beans and connectors allows for aspect reusability since aspect beans describe crosscutting concerns independently with the concrete component context while connectors deploy aspect beans into concrete components. Hence, the combination of AOSD and CBSD in JAsCo does not only brings AOP support to component-based applications but also adopt component-based ideas to the AOSD approach.

In JAsCo, aspects are deployed within an application using *connectors*. Assume that the application consists of a *Printer* component which needs the access-control functionality implemented by the above aspect. What we can do is to bind all executions of method *printFile* from the *Printer* package to the *AccessControl* hook of the *AccessManager* class using a connector as follows:

```
1   connector printAccessControl{
        AccessManager.AccessControl control =
3           new AccessManager.AccessControl(void Printer.printFile(file));

5       control.setExceptionMessage("Printer Access Exception");
        control.around();
7   }
```

The above connector namely *printAccessControl* consists of a hook initialisation (line 2-3) that deploys the *AccessControl* hook represented by *control* to the *printFile* method. This hook initialisation is similar to a Java class initiation where *control* is initialised with method signatures just as previously specified in the hook constructor. Next, the connector indicates that the *setExceptionMessage* method and the *around* method of the control will be executed(line 5-6) whenever method *printFile* of the *Printer* component is called. Note that JAsCo provides some supports for the declaration of precedence and combinations strategies in connectors in order to make explicit how aspects are applied in case there are conflicts among aspects.

JAsCo supports the specification of the applicability of aspects in terms of a sequence or a protocol [125]. This feature is implemented based on the generic framework proposed by Dounce et al. previously described in section 3.2.2.1. Let us consider an example of stateful aspects in JAsCo taken from [125]. The previous *AccessControl* aspect is now extended to a new aspect called *StatefulAccessControl*. The following code shows the specification of that new aspect.

```
1   class StatefulAccessManager extends AccessManager{
        hook StatefulAccessControl{
3           StatefulAccessControl(starttrigger(..a1), method(..a2), stoptrigger(..a3))){
                start > p1;
5               p1: execution(starttrigger) > p3||p2;
                p2: execution(method) > p3||p2;
7               p3: execution(stoptrigger) > p1;
            }
9
            around p2(){
11              if(pdb.check(currentuser,thisJoinPointObject))
                    return proceed();
13              else
                    throw new AccessException(exceptionmessage);
15          }
        }
17  }
```

The constructor of the aspect has three abstract method parameters as input: *starttrigger, method* and *stop* (line 3). The constructor body specifies the stateful pointcut that can be interpreted as a transition system (line 4-7). Each transition is described by a name (*e.g., p*1), a pointcut (*e.g., execution(starttrigger)*), and one or more destination transitions (*e.g., p*3 or *p*2) to be matched after the current transition is fired. In the above example, there is an advice applicable to transition *p*2 (line 14-18). This advice will be triggered when transition *p*2 defined in the hook constructor is fired.

In JAsCo, aspects can be added and removed at run-time. The implementation of JAsCo language features, especially JAsCo dynamic features is based on the Jutta aspectual just-in-

time compiler and the HotSwap run-time byte-code instrumentation framework [120].

3.4.4 CaesarJ

CaesarJ [91, 23] is an aspect-oriented programming language that is based on object-oriented concepts and particularly integrates with the Java language. CaesarJ aims at better modularity for aspects and components. In short, CaesarJ supports reusable aspects and non-invasive integration mechanisms for components. In the following, we present a few major features of the CaesarJ language.

The CaesarJ model supports the notion of a *class family* [51], a large-scale unit of modularity which involves a group of related classes. Abstraction, late binding, and subtype polymorphism are also supported at the level of class families. Furthermore, CaesarJ supports the concept of *virtual classes*. A virtual class is an abstraction of an inner class of an enclosing class family. Just like a virtual method, a virtual class may have different meanings depending on the dynamic context when it is used. The following code [23] represents the base class family *HierarchyDisplay* as well as two extensions *AdjustedHierarchyDisplay* and *AngularHierarchyDisplay*.

```
1   cclass HierarchyDisplay{

3       cclass Node {...}

5       cclass CompositeNode extends Node {
            ...
7           calculateLayout() {
                ...
9               Connection c = new Connection();
                ...
11          }
        }
13
        cclass Connection {...
15          void initShape(Point pt) {...}
        }
17
        Node root;...
19  }

21  cclass AdjustedHierarchyDisplay extends HierarchyDisplay{
        cclass Node {...
23          int maxwidth;
        }
25
        void foo(Node n) {... n.maxwidth...}
27  }

29  cclass AngularHierarchyDisplay extends HierarchyDisplay {
        cclass Connection {...
31          void initShape(Point pt) { ...}
        }
33  }
```

The *HierarchyDisplay* class family involves three inner classes *Node*, *CompositeNode*, and *Connection*. All these classes are described as CaesarJ virtual classes using keyword `cclass` in order to differentiate them from pure Java classes. The *AdjustedHierarchyDisplay* class (line 21-24) is defined as an extension (subclass) of the *HierarchyDisplay* class. Its definition shows that the *Node* inner class is refined from that of the superclass and a new method called *foo* is added. Note that all references to type *Node* in the other classes of the class family are automatically rebound to type *Node* of the *AdjustedHierarchyDisplay* class during an execution of an object of type *AdjustedHierarchyDisplay*. This is the fundamental difference between binding in CaesarJ virtual classes and pure Java classes. Furthermore, CaesarJ supports a composition operator in order to compose two classes. For instance, we can define a new class called *AdjustedAngularHierarchyDisplay* by composing two classes *AdjustedHierarchyDisplay* and *AngularHierarchyDisplay* as follows:

```
1   cclass AdjustedAngularHierarchyDisplay extends
                AdjustedHierarchyDisplay & AngularHierarchyDisplay{}
```

CaesarJ adopts the join point interception model from AspectJ in order to modularise crosscutting concerns. One important feature of CaesarJ in comparison with AspectJ is that aspects in CaesarJ are designed as components and thus they are more independent and easier to be reused. The following code [23] illustrates a class namely *CompanyLogger* that contains a pointcut and an advice in order to perform tracing tasks:

```
    deployed public cclass CompanyLogger{
2       pointcut logMethods(): execution(* company.*.*(..)) || execution(company.*.new(..));
        before(): traceMethods() {
4           System.out.println(thisJoinPointStaticPart.toString());
        }
6   }
```

CaesarJ supports compile-time as well as runtime activation of an aspect. It enables flexible control over activation and scope of aspects by providing different control mechanisms over aspects from outside. CaesarJ supports the definition of *wrapper* classes and automated wrapper recycling. A wrapper can introduce new fields and methods to other classes (*wrappees*). Wrappers are used as adapters of existing classes in order to map between types from two domains.

3.4.5 Summary

In this section, we have discussed a few major approaches on aspects and components. All of those approaches basically support necessary language constructs for the definition of aspects in component-based applications. However, while interaction protocols shown to be useful for component-based applications, most existing approaches on AO over components do not consider specific language supports for describing aspects over component protocols. Furthermore, most existing AO approaches for components do not consider the problem of evolution on components resulting from applying advice changes the desirable properties of the original system.

3.5 Conclusions

As we have learned from the number of work on AOP, in general, an aspect-oriented language for component-based applications should satisfy the following requirements:

- **Language expressiveness:** The language used to define the patterns of crosscutting concerns should be expressive enough to capture as much patterns as possible, especially, sequences of events (which are very common in applications). However, note that, there is always a trade-off between expressiveness and analysis support. Therefore, language expressiveness should be only pushed to a certain limit where a balance between the ability to define patterns and the requirement for doing some kinds of analysis to ensure correctness can be reached.

- **Analysis support:** The ability of language to support analysis is important to ensure correctness for software. As we have just mentioned above, there is a trade-off between language expressiveness and analysis support. A survey on history-based aspect languages show that some analysis can be done for the regular-based ones. The advances in language theory research may help to attain analysis support for more expressive aspect-oriented languages. In addition to the own ability of language on analysis support, we also need more verification methods dedicated for aspect-oriented programs.

- **Mechanisms for interaction detection and resolution:** The parallel composition of aspects may lead to conflicts since they are supposed to be developed independently of each other. With the help of the ability to support analysis of the aspect-oriented language, we could develop algorithms to detect potential interactions between aspects and should provide mechanisms for conflict resolution in situations where an interaction may cause undesirable effects.

- **Mechanisms for the specification and verification of interface between aspects and base programs:** It would be useful to have a way for aspects to declare the set of requirements they need as pre-conditions for the base programs to satisfy in order to ensure they can produce expected effects. Moreover, this ability also helps to improve aspect reuse. On the contrary, the availability of an abstraction about the base program may help to reduce the cost of analysis on aspects and the base program.

- **Specific support for component-based applications:** Since protocols are widely used in component software, support for the declaration and composition of protocols is indispensable for an aspect-oriented language that gears towards component-based systems. Furthermore, analysis methods for critical properties of component-based systems such as compatibility and substitutability should be available.

We have concluded the part which focuses on State of the Art in this thesis. We will next move to the part on Contributions in which we present our proposal for an aspect-oriented language for component-based systems and our approaches on analysis and verification for the language we have developed.

Part II

Contributions

Chapter 4

VPA-based Aspect Language

This chapter presents the main contribution of the thesis: a new history-based aspect language defined using visibly pushdown automata (VPAs), the VPA-based Aspect Language (VPAL). We will use the abbreviated name VPAL to refer to our aspect language from now on. This chapter is organized as follows. Section 4.1 motivates the use of visibly pushdown automata to define history-based aspect languages. Section 4.2 presents the formal definition and important properties of the VPA and the class of visibly pushdown languages described by these VPAs. Section 4.3 presents the formal syntax and semantics of VPAL. Section 4.4 introduces the library that we have implemented in order to provide supports for property analysis on VPA-based aspects. Section 4.5 shows how interactions among VPA-based aspects can be detected. Finally, Section 4.6 concludes the chapter.

4.1 Motivation

In Section 2.4, we have discussed the important role of interaction protocols in component-based software. Integrating interaction protocols into component interfaces provides more information on the component behavior and thus enables more precise property analysis. There have already been a number of aspect-oriented languages that specially aim at component-based software as discussed in Section 3.4. However, most of them do not support protocol-based pointcut languages.

In Section 3.2.2, we have reviewed a few history-based aspect languages that exploit protocol-based pointcut languages in order to enable declarative aspect definitions and, for some of them, support reasoning over properties of aspect-oriented programs. The pointcut languages supported by those approaches can be divided into two categories: regular and non-regular ones. In general, more approaches feature regular pointcut languages than non-regular ones. Approaches featuring regular pointcuts have shown that they have an important characteristic that existing non-regular ones typically do not have: they support automatic property analysis as analysis problems for regular languages have been well understood theoretically and practical solutions exist. However, the limitation of AO languages that feature regular pointcuts lies in the restricted expressivity of regular languages. For example, they can not describe protocols that require a memory such as recursive protocols or nested sequences. Unfortunately, there are quite a large number of applications that involve such types of protocols.

Although aspect languages with non-regular pointcuts can express more types of proto-

cols in a pointcut declaration than those featuring regular pointcuts, they have two major limitations. First, no support for property analysis is available because of the lack of effective analysis techniques for non-regular languages. Second, due to high expressivity of such languages, it is difficult to obtain efficient implementations for those languages.

Let us take a basic peer-to-peer query protocol as an example to demonstrate the limitations of existing regular and non-regular aspect languages.

Example: a peer-to-peer query protocol. A peer-to-peer network denotes a distributed architecture in which equally privileged peers with equal resources cooperate to solve tasks. The protocol over peers that we are interested in describes the abstract control flow of a query algorithm over peer nodes. Assume that every peer in the network connects to at least one other peer (neighbor). A peer initiates a query by sending query messages to all of its neighbors. If a receiving peer does not have the answer for the query, it forwards the query to its neighbors. For simplicity, let us just assume that cycles (in which a query comes back to a peer node it has already passed by) will be detected and handled. If a receiving peer has the answer it sends a reply message to the peer from which it received the query message. Eventually, the answer can be propagated back to the original peer that initiates the query.

Now we wish to apply an aspect that monitors the query process and notifies the server when all the replies to the initiating queries have been received. Using the aspect language that supports regular pointcuts proposed by Douence et al. [48], we could implement (or better approximate) the above monitoring aspect as follows:

$$A_{monitoring} = \mu a.(query \quad | \quad reply \rhd notify) * \ ; \ a$$

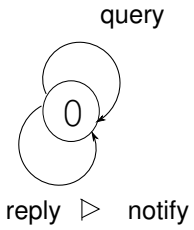


Figure 4.1: Monitoring aspect implemented using regular language

Figure 4.1 illustrates the above aspect which admits, among others, sequences of alternative calls to query and reply and triggers *notify* events. However, it does not capture only the final replies (*i.e.,* replies to the queries directly sent by the first node). Instead, it notifies the server whenever a *reply* event occurs. Furthermore, the aspect does not detect situations in which the number of replies exceeds the number of queries, because the regular query language is not capable of capturing properly nested structures. Replacing the or-expression in the recursion by the sequence *query;reply* does not work either: the expression just match flat and not nested sequences. While we can obviously use aspect variables to capture the matching replies, the resulting system is not amenable to property analysis performed at the protocol level. For instance, analyzing whether the base program satisfies the aspect requirements or whether two aspects interact based on their pointcuts cannot be done by just checking the automata representing the aspects and/or the base programs because there of the variables involved and the code manipulating them.

The monitoring aspect could be implemented more properly using an aspect language that supports non-regular, Turing-complete or not, pointcuts. Although Turing-complete languages are the most expressive languages they are not widely used for modeling protocols in software applications because of their very limited support for any kind of property analysis. On the other hand, non-regular but not Turing-complete protocols that can express recursion or properly nested structures are commonly used in many applications especially the peer-to-peer application domain. One particular type of non-regular protocols that has been used are those defined by pushdown automata, *e.g.,* [127].

A pushdown automaton (PDA) is essentially an extension of a finite state automaton (FSA) with the addition of a stack data structure [67]. During an operation of a pushdown automaton, transitions are chosen based on the input symbol, the current state, and the symbol at the top of the stack. Furthermore, the stack of a PDA may be optionally manipulated, *i.e.,* symbols can be pushed to the stack or popped from the stack, when transitions are taken. Hence, in comparison to a FSA, a PDA is equipped with an extra memory that can be used to store some information along its operation that enables context-free languages to be recognized.

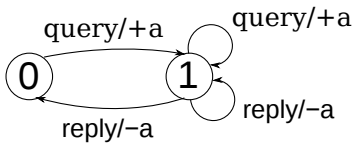


Figure 4.2: Modeling query protocol using PDA

Using a PDA, we can properly model the query protocol as shown in Figure 4.2. The PDA in figure 4.2 consists of two states and four possible transitions representing query and reply events. When a query transition is taken, an a symbol is pushed to the stack. When a reply transition is taken, an a symbol is popped from the stack. If there is no a symbol on the top of the stack then a reply transition cannot be taken. Consequently, modeling the query protocol using a PDA ensures that the number of replies does not exceed the number of queries. Furthermore, the last reply, which is the reply transition from state 1 to state 0, can be detected. Hence, modeling the query protocol using a PDA describes the protocol much more precisely than using a FSA, *i.e.,* a regular language.

The class of context-free languages, which is characterized by PDAs, is employed as a basis for the definition of compiler frontends for most programming languages nowadays because they support the definitions of nested program modules and recursive function calls. An aspect language featuring context-free-based (or PDA-based) pointcuts could capture the above query protocol properly and corresponding approaches have been proposed, see Section 3.2.2.2. However, many interesting properties of context-free languages and programs cannot be analyzed statically. For example, checking whether a context-free language is included in another one is undecidable. To the best of our knowledge, no aspect language has been proposed that can both express such protocols and support property analysis. For instance, the example aspect above can be expressed using the context-free aspect language introduced by Walker and Viggers [128]. However analyzing whether the base program satisfying the aspect requirements or whether two aspects interact is not supported. Furthermore, there has been no efficient implementation proposed for this aspect language.

We aim at developing a language that can define aspects over these protocols properly

while still be amenable to some meaningful property analysis. We are furthermore interested in corresponding implementation techniques and efficient analysis techniques, *e.g.,*, by the reuse of existing model checkers through approximation of the non-regular protocols by (typically complex) regular expressions.

Visibly pushdown languages (VPLs), introduced by Alur and Madhusudan in 2004 [21] and defined using visibly pushdown automata (VPAs), are a good candidate to meet the aforementioned requirements. Two important reasons for this choice are that VPAs can express many protocols which can only be expressed by context-free languages (and not regular languages) and that VPLs, in contrast to context-free languages, satisfy most properties of regular languages.

4.2 Visibly Pushdown Automata

As discussed in the previous section, the main limitation of context-free languages for our purposes consists in their lack of support for property analysis. There are, however, algorithmic solutions for certain classes of non-regular languages have been put forward, *e.g.,* [35], [52], [70]. The common characteristic of these approaches is that the stack is explicit. Alur and Madhusudan [21] have developed this observation into a new kind of automaton, called visibly pushdown automaton (VPA). The class of visibly pushdown languages (VPLs) defined by VPAs is of similar expressivity as context-free languages but still tractable as regular languages. We have chosen this language class as a foundation for the development of our aspect language due to its promising capabilities in expressivity and support for property analysis.

Since we later present arguments and proofs of correctness involving our aspect language in terms of the definitions underlying VPAs, we present in this section the definition and key characteristics of this language and automata class. We summarize definitions, closure properties and decision properties of VPAs from their original presentation by Alur and Madhusudan [21] (see this paper also for the proofs of the properties mentioned in this section.).

4.2.1 Definitions

VPAs are basically a restricted form of PDAs. Technically, the input alphabet $\widetilde{\Sigma}$ of a visibly pushdown automaton is partitioned into three disjoint sets: $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ where Σ_c is a finite set of *calls*, Σ_r a finite set of *returns*, and Σ_l a finite set of *local actions*. Intuitively, a visibly pushdown automaton pushes onto the stack only when it reads a call, pops the stack only when it reads a return, and does not change the stack when it reads a local action. The notion of visibility of stack-related actions thus consists in restricting each of them to one of the three partitions.

A visibly pushdown automaton can be formally defined as follows:

Definition 1 (Visibly pushdown automaton)**.** *A (non-deterministic) visibly pushdown automaton M on finite words over $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ is a tuple*

$$M = (Q, Q_{in}, \Gamma, \delta, Q_F)$$

where

[0]The reader can skip this section if she/he is already familiar with the class of Visibly Pushdown Automata or she/he prefers to learn about this class from the original paper [21].

- Q *is a finite set of states*
- $Q_{in} \subseteq Q$ *is a set of initial states*
- Γ *is a finite stack alphabet that contains a special bottom-of-stack symbol* $\perp$
- $\delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \backslash \{\perp\})) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_l \times Q)$ *is a finite set of transitions*
- $Q_F \subseteq Q$ *is a set of final states*

According to the above definition, given two states $q, q' \in Q$, a transition $t \in \delta$, where the control changes from q to q', can be expressed in one of the three formats:

- $t = (q, a, q', \gamma)$: t is a push (or call) transition where $a \in \Sigma_c$ and $\gamma \in \Gamma \backslash \{\perp\}$, *i.e.,* on reading a, γ is pushed to the stack.
- $t = (q, a, \gamma, q')$: t is a pop (or return) transition where $a \in \Sigma_r$ and $\gamma \in \Gamma \backslash \{\perp\}$, *i.e.,* on reading a, γ is popped from the stack if it is the symbol on the top of the stack.
- $t = (q, a, q')$: t is a local transition and the stack is not manipulated.

Note that ϵ-transitions are not allowed in a VPA.

Run. For a word $w = a_0 a_1 ... a_k$, a *run* of M on w (where $a_0, a_1, .., a_k$ are inputs to M in that order) is a sequence $\rho = (q_0, \sigma_0), (q_1, \sigma_1), ..., (q_k, \sigma_k)$ where $q_i \in Q$, σ_i is the stack content at state q_i, $\sigma_0 = \perp$ and there is a transition from q_i to q_{i+1}. Note that, a run ρ is an accepting run of M on w if the last state is a final state *i.e.,* $q_k \in Q_F$. Hence, acceptance of VPAs is defined by reaching final states, not by stack emptiness.

Language. The *language* of M, denoted as $L(M)$, is the set of all words $w \in \Sigma*$ accepted by M. We can define L(M) as follows:

$$L(M) = \{w | (q_0, w, \perp) \vdash_M^* (q, \epsilon, \sigma_k)\}$$

where $q \in Q_F$ and σ_k is the stack content at state q. That is, starting in q_0, M consumes w from the input and eventually enters an accepting state. The $\vdash_M^*$ operator indicates a sequence of several transitions on M.

Definition 2 (Visibly Pushdown Languages)**.** *A language L is a visibly pushdown language with respect to a partition $\widetilde{\Sigma}$ if there is a VPA M over $\widetilde{\Sigma}$ such that $L(M) = L$.*

Note that, on the one hand, every regular language is also a VPL but not vice versa. On the other hand, every VPL is a context-free language obtained simply by merging the input symbol partitions. For instance, if $\Sigma_c = \{a\}$ and $\Sigma_r = \{b\}$ then the language $a^n b^n (n \geq 0)$ is a VPL but the language $b^n a^n$ is not a VPL w.r.t this partitioning. Hence, protocols that involve properly nested structures can be expressed using VPLs with appropriate partitioning. For instance, the query protocol from the beginning of this section can be defined by a VPA M similar to the PDA on figure 4.2 with a partitioned input alphabet $\widetilde{\Sigma} = \langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ where $\Sigma_c = \{query\}, \Sigma_r = \{reply\}, \Sigma_l = \emptyset$.

4.2.2 Closure properties of Visibly Pushdown Languages

In this section we present the major closure properties of VPLs and discuss how they can be proved. These properties are very important, especially intersection and complementation, because they underlie property analysis of VPA-based protocols.

Theorem 1 (Closure)**.** *Let L_1 and L_2 be visibly pushdown languages with respect to $\widetilde{\Sigma}$. Then, $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1.L_2$, and L_1^* are visibly pushdown languages with respect to $\widetilde{\Sigma}$.*

Like regular languages, VPLs are closed under union, intersection, complement, concatenation and the Kleene-star operation. Note that, context-free languages are not closed under intersection and complementation. In the following we discuss these closure operations and properties in more details.

Closure Under Union. Considering L_1 and L_2 are two VPLs accepted by two VPAs $M_1 = (Q_1, Q_{in_1}, \Gamma_1, \delta_1, Q_{F_1})$ and $M_2 = (Q_2, Q_{in_2}, \Gamma_2, \delta_2, Q_{F_2})$ respectively, the language $L = L_1 \cup L_2$ is accepted by a VPA $M = M_1 \cup M_2$ defined as follows:

$$M = (Q_1 \cup Q_2, Q_{in_1} \cup Q_{in_2}, \Gamma_1 \cup \Gamma_2, \delta_1 \cup \delta_2, Q_{F_1} \cup Q_{F_2})$$

i.e.,, the set of states, of transitions, initial states, and final states of M are the pointwise unions of the corresponding argument sets.

Closure Under Concatenation. Given an input word w that can be split into w_1, w_2 where $w_1 \in L_1, w_2 \in L_2$, a VPA M that accepts $L_1.L_2$ simulates w_1 on M_1 and w_2 on M_2 using different stack alphabet, and when simulating M_2 the stack alphabet of M_1 is treated as bottom of stack.

Closure Under Intersection. The intersection of two VPAs M_1, M_2 w.r.t the same partition $\widetilde{\Sigma}$ is defined by a VPA M as follows:

$$M = (Q_1 \times Q_2, Q_{in_1} \times Q_{in_2}, \Gamma_1 \times \Gamma_2, \delta_1 \times \delta_2, Q_{F_1} \times Q_{F_2})$$

Closure Under Kleene star. A VPA $M*$ that accepts $L*$ simulates M step-by-step but it can non-deterministically restart M when M is in a final state. When this happens, the actual stack content is treated as empty.

Theorem 2 (Determinization)**.** *For any VPA M over $\widetilde{\Sigma}$, there is a deterministic VPA M' over $\widetilde{\Sigma}$ such that $L(M) = L(M')$. Moreover, if M has n states, we can construct M' with $O^{2^{n^2}}$ states with stack alphabet of size $O(2^{n^2}.|\Sigma_c|)$.*

Given the same input partition $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ we can construct a deterministic VPA $M' = (Q', Q'_{in}, \Gamma', \delta', Q'_F)$ from a nondeterministic VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ where

$$\begin{aligned}
&Q' = 2^{Q \times Q} \times 2^Q \\
&Q'_{in} = \{(Id_Q, Q_{in}) \mid Id_Q = \{(q,q) | q \in Q\}\} \\
&\Gamma' = \{(S, R, a) \mid (S, R) \in Q', a \in \Sigma_c\} \\
&Q'_F = \{(S, R) \mid R \cap Q_F \neq \emptyset\}
\end{aligned}$$

and the transition relation δ' is defined as follows:
(Local) For every $a \in \Sigma_l$, $((S,R),a,(S',R')) \in \delta'$ where $S' = \{(q,q') \mid \exists q'' : (q,q'') \in S, (q'',a,q') \in \delta\}, R' = \{q' \mid \exists q \in R : (q,a,q') \in \delta\}$.
(Call) For every $a \in \Sigma_c$, $((S,R),a,(Id_Q,R'),(S,R,a)) \in \delta'$ where $R' = \{q' \mid \exists q \in R, \gamma \in \Gamma : (q,a,q',\gamma) \in \delta\}$.
(Return)

- For every $a \in \Sigma_r$, $((S,R),a,(S',R',a),(S'',R'')) \in \delta'$ if (S'',R'') satisfies the following: Let $Update = \{(q,q') \mid \exists q_1,q_2 \in Q, \gamma \in \Gamma : (q,a',q_1,\gamma) \in \delta, (q_1,q_2) \in S, (q_2,a,\gamma,q') \in \delta\}$. Then, $S'' = \{(q,q') \mid \exists q_3 : (q,q_3) \in S', (q_3,q') \in Update\}$ and $R'' = \{q' \mid q \in R', (q,q') \in Update\}$.

- For every $a \in \Sigma_r$, $((S,R),a,\bot,(S',R')) \in \delta'$ if $S' = \{(q,q') \mid \exists q'' : (q,q'') \in S, (q'',a,\bot,q') \in \delta\}, R' = \{q' \mid \exists q \in R : (q,a,\bot,q') \in \delta\}$.

In the following we briefly explain the above definition of determinization[1]. Basically, the construction of an equivalent deterministic VPA from a non-deterministic one is similar to a subset construction [67] on a non-deterministic FSA. However, since there is a stack associated with the VPA, handling push transitions that manipulate the stack is postponed until the handling of their corresponding pop-transitions. A set S in the above definition represents the set of "summary" edges, which includes all transitions that occur from a push transition to its corresponding pop-transition. A set R represents the set of reachable states. Hence, instead of performing a subset construction on the set of states, we perform a subset construction on the set of summary-edges.

Closure Under Complementation. The class of visibly pushdown languages is closed under complementation. That is, if L is a visibly pushdown language over partitioned alphabet $\widetilde{\Sigma}$ then $\overline{L}$ is also a visibly pushdown language over $\widetilde{\Sigma}$.

We can define a VPA $\overline{M}$ which is the complementation of a VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ as follows:

$$\overline{M} = (Q, Q_{in}, \Gamma, \sigma, Q - Q_F)$$

That is, $\overline{M}$ is exactly like M but the accepting states of M have become non-accepting states of $\overline{M}$ and vice versa. M must be a deterministic VPA since otherwise a given input word might end in both a final and a nonfinal state. In this case the word should be rejected by $\overline{M}$, but inverting the final and nonfinal states would result in it being accepted (because it will also end in both a final and a nonfinal state of $\overline{M}$).

Unlike for PDAs, it is possible to construct a deterministic VPA from a non-deterministic using the determinization operation introduced above.

4.2.2.1 Decision properties of Visibly Pushdown Languages

In the following, we discuss three decision properties of VPLs: emptiness, universality, and inclusion. These properties also play an important role for solving analysis problems.

[1]The ideas were initially presented in the original paper [21]. However, there is no significant text that is copied from there.

Emptiness. The emptiness problem is to check whether the language defined by a VPA is empty. Emptiness of a VPA is decidable in time $O(n^3)$ where n is the number of states in the VPA.

Inclusion. The inclusion problem is to check whether given two VPAs M_1 and M_2, $L(M_1) \subseteq L(M_2)$. This problem is decidable for VPLs in EXPTIME-complete. Given VPAs M_1, M_2, we can take the complement of M_2, take its intersection with M_1 and check for emptiness. Note that the inclusion problem is undecidable for context-free languages.

Universality. The universality problem is to check whether a given VPA M accepts all strings in Σ^*. This problem is decidable in EXPTIME-complete. This problem can be reduced to checking the inclusion of the language of the fixed 1-state VPA M_1 accepting Σ^* with the given VPA M. Note that the universality problem is undecidable for context-free languages.

Although the complexity of these above computations is still high, closure and decision properties of VPLs theoretically enable many types of property analysis that are used to be impossible with context-free languages. We will discuss how we apply them to our analysis problems with the aspect language later on. We furthermore show that an optimized implementation of the operations yields reasonable runtime and space requirements for practically relevant VPA-based protocols.

4.3 VPA-based Aspect Language

In this section, we present the VPA-based aspect language we have developed. This aspect language essentially extends the framework of regular aspect languages proposed by Douence et. al [47] by means for the formulation of VPA-based pointcuts and advice. It features, in particular, constructors for the declarative definition of pointcuts based on regular and non-regular structures and advice mechanisms for the manipulation of behaviors that require balanced matching

In the following, we will first give an overview of the language then we introduce the syntax of the language and use a set of small examples in the context of a peer-to-peer application to demonstrate its features. Next, we present its formal semantics in terms of a small-step semantics that can serve as a foundation for a (future) implementation of the language.

4.3.1 Overview

As in most other aspect languages, three basic mechanisms form VPA-based aspects: aspects, pointcuts, and advice.

- Aspects are defined either as atomic aspects (which cannot be split further into simpler aspects) or composite aspects (which are the composition, such as a sequence, of several aspects).

- Pointcuts permit the matching of event sequences that can be described by VPAs. Pointcuts can be defined as single terms which represent individual events. The terms are categorized into three types corresponding to three transition types of a visibly push-down automaton: calls, returns, and locals. Furthermore, pointcuts can be defined by

two specific constructors: depth constructors that define nested sequences whose depths satisfy certain conditions, and the permutation constructors that define sequences of permutations of pairs of events. Pointcuts also include forms for regular expressions for convenience (although regular expressions are a subset of those expressible using VPAs).

- The advice class permits the definition of advice as sequences of events, or the specific advice operator *closeOpenCalls*. The latter permits the introduction of missing return statements.

Not all aspects defined according to the provided language syntax are valid: some return events in a pointcut, for instance, may not correspond to appropriate call events. We believe that a restricting the language by, for instance, closely coupling calls and returns is not desirable for an aspect programming language. However, later we investigate subsets of the permissable sentences of the language that ensure their validity, for instance, syntactically.

4.3.2 Syntax

Figure 4.3 presents the grammar defining the syntax of VPA-based aspects. In this figure, non-terminals are set in italic type. Terminals are informally introduced using the comment symbol '//'. Keywords are set in bold face type. Repetitions are marked using parentheses (*e.g.,*.$Term\{,Term\}$ expresses that the expression may be defined by a sequence of several terms separated by the ',' symbol). Lexical categories are marked using all uppercase letters.

In the following we give a detailed explanation on three major language constructs that are used to define a VPA-based aspect: *aspect, pointcut* and *advice*.

4.3.2.1 Aspects

Aspects are defined using the first six rules of the grammar. The first and second rule define an atomic aspect that comprises a pointcut and an advice, just like aspects defined in many other traditional aspect languages. The left part of the ' $\triangleright$ ' operator always represents the pointcut and the right part of the ' $\triangleright$ ' operator represents the advice of an aspect. For instance, the following aspect:

$$login \triangleright CreateLog$$

describes that $CreateLog$ (the advice) would be executed if there is an invocation to *login* (the pointcut). Note that it is possible for an aspect to be defined only with a pointcut. In this case the $\triangleright$ operator and the (empty) advice are omitted from the pointcut expression. This form is useful in compositions, such as sequence aspects.

The third rule defines a composite aspect that can be formed as a sequence of atomic aspects. For instance, according to the following composite aspect:

$$login \triangleright CreateLog \ ; \ query \triangleright WriteLog$$

$CreateLog$ will be executed when a call to *login* occurs then $WriteLog$ on occurrence of a call to *query*.

Sequences '$a_1 \ ;_{Tag} \ a_2$' may be restricted by specifying (call, return or local) events, see non-terminals Tag and $Term$, that must not occur between the occurrences of a_1 and a_2. For example, assume that the base program in the above example may invoke event *logout*

$$
\begin{array}{lll}
\mathit{Aspect} & ::= & \mathit{Pointcut} \\
& | & \mathit{Pointcut} \rhd \mathit{Advice} \\
& | & \mathit{Pointcut} \rhd \mathit{Advice} \ ;_{\mathit{Tag}} \ \mathit{Aspect} \\
& | & \mu \ \mathrm{ID} \ . \ \mathit{Aspect} \\
& | & \mathit{Pointcut} \rhd \mathit{Advice} \ ;_{\mathit{Tag}} \ \mathrm{ID} \\
& | & \mathit{Aspect} \ \Box \ \mathit{Aspect} \\
\\
\mathit{Tag} & ::= & \mathit{Term}\{,\mathit{Term}\} \ \mid \ \neg \ (\mathit{Term}\{,\mathit{Term}\}) \ \mid \ \epsilon \ \mid \ \emptyset \\
\\
\mathit{Pointcut} & ::= & \mathit{Term} \ \mid \ \mathit{DepthOp} \ \mid \ \mathit{PermutationOp} \ \mid \ \mathit{RegEx} \\
\mathit{RegEx} & ::= & \mathit{Term} \\
& | & \mathit{RegEx} \ ;_{\mathit{Tag}} \ \mathit{RegEx} \ \mid \ \mathit{RegEx} \| \ \mathit{RegEx} \ \mid \\
& & \mathit{RegEx} \ [\ \mathrm{CONSTANT} \ ] \ \mid \ \mathit{RegEx} \ + \\
\\
\mathit{Term} & ::= & \mathit{Term}(\mathit{ARGS}) \ \mid \ \mathit{Local} \ \mid \ \mathit{Call} \ \mid \ \mathit{Return} \\
\mathit{Local} & ::= & \mathrm{ID} \\
\mathit{Call} & ::= & \mathrm{ID}_{\mathit{StackSymbol}} \\
\mathit{Return} & ::= & \underline{\mathrm{ID}_{\mathit{StackSymbol}}} \\
\mathit{StackSymbol} & ::= & \overline{\mathrm{ID}} \\
\\
\mathit{DepthOp} & ::= & \mathbf{depth}^{\mathrm{CONSTANT}}_{\mathit{Call},\mathit{Return}} \ \mid \ \mathbf{depth}^{\leq \mathrm{CONSTANT}}_{\mathit{Call},\mathit{Return}} \ \mid \ \mathbf{depth}^{\geq \mathrm{CONSTANT}}_{\mathit{Call},\mathit{Return}} \\
\mathit{PermutationOp} & ::= & \mathbf{fp}([\mathit{Call},\mathit{Return}]\{,[\mathit{Call},\mathit{Return}]\}) \\
& | & \mathbf{np}([\mathit{Call},\mathit{Return}]\{,[\mathit{Call},\mathit{Return}]\}) \\
\\
\mathit{Advice} & ::= & \mathit{Term} \ \{; \mathit{Term}\} \\
& | & \mathbf{closeOpenCalls}_{\mathit{Return}} \\
& | & \mathit{Term} \ [\ \mathrm{CONSTANT} \ ] \\
\\
\mathrm{ID} & & // \ \textit{identifiers} \\
\mathrm{CONSTANT} & & // \ \textit{constant numbers} \\
\mathrm{ARGS} & & // \ \textit{variable names}
\end{array}
$$

Figure 4.3: Grammar for a VPA-based aspect

and then invoke *query*. In this case, since the aspect is not prepared to handle the *logout* event, it just ignores this event and proceeds to match the *query* event and execute $WriteLog$ when the *query* event occurs. To avoid this potentially problematic situation, we can exclude *logout* events from matching as follows:

$$login \rhd CreateLog \;;_{logout}\; query \rhd WriteLog$$

Note that the sequence operator ';' without a Tag value behaves as the sequencing operator that is commonly used also in other history-based aspect languages, that is, no events are excluded from matching.

The fourth and fifth rules enable the expression of repetition in an aspect via tail recursion. An aspect defined as '$\mu a.A$' where a is a variable and A is an aspect is equivalent to the aspect A where all the occurrences of a are replaced by $\mu a.A$. The last rule defines an aspect as a choice of two aspects using the '$\Box$' operator. Let us consider an example of an aspect defined using these three rules:

$$\begin{array}{l} \mu a_1.\ login \rhd CreateLog; \\ \quad \mu a_2.\ logout \rhd CloseLog; a_1 \\ \quad\quad \Box\ query \rhd WriteLog; a_2 \end{array}$$

According to the above definition, when a call to *login* is detected from the base execution, the aspect triggers advice $CreateLog$ then it waits for two alternative incoming events: *logout* or *query*. If the aspect encounters *logout* first, it triggers the advice $CloseLog$ and comes back to the point where it looks for the *login* event as specified by the repetition variable a_1. Otherwise, if the aspect encounters *query*, it triggers the advice $WriteLog$ and comes back to the point where it waits for *logout* or another *query* event as specified by the variable a_2.

4.3.2.2 Pointcuts

There are four different forms of pointcuts for VPA-based aspect: a single term, a depth-testing VPA-specific pointcut constructor, a constructor allowing to match for permuted events and a regular expression pointcut.

Terms. A term is a method call with arguments that is explicitly categorized according to the partitioning of input symbols in VPAs: *local* transitions that may not influence the stack, *call* transitions that push on the stack and *return* transitions that pop the top of the stack. A term representing a call transition is tagged with the stack symbol that will be pushed on the stack. A term representing a return transition, which we underline in order to differentiate it from call transitions, is also tagged with the stack symbol that should be available on top of the stack when the transition is taken. This explicit transition classification enables the definition of VPA-based non-regular pointcuts that can capture well-balanced contexts.

Let us consider an aspect that triggers a call to an *abort* function when there is an *abortRequest* sent from a peer in a peer-to-peer query protocol. Figure 4.4 illustrates the query protocol.

The aspect that implements the abort function is defined using our VPALas follows:

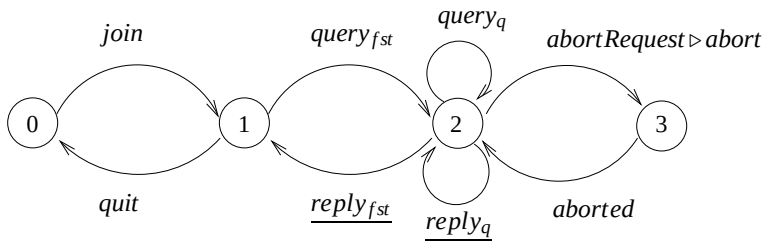


Figure 4.4: Aspect handling abortRequest in peer-to-peer query protocol

$$join\ ;\ \mu b.\ query_{fst}\ ;\ \mu c.(\underline{reply_{fst}}\ ;\ b) \\ \square\ (query_q \square \underline{reply_q}\ ;\ c) \\ \square\ (abortRequest \triangleright abort\ ;\ c)$$

Note that there is one requirement for this aspect. That is, an *abortRequest* message should only be treated by the aspect if there is at least one on-going query. This is a reasonable constraint if the abort function is an operation which might consume much time or other resources. Since it is possible to tag the call and return transitions with stack symbols, we can easily differentiate the first query from the other ones and thus can express the condition where *abortRequest* should be treated by the aspect. Furthermore, tagging stack symbols to the transition also ensures the number of replies does not exceed the number of query during matching process.

Depth constructors. Three depth constructors enable aspect to be defined in terms of the number of (typically matching) calls and returns that have occurred. The constructor $\mathbf{depth}^{n}_{m,\underline{o}}$ defines a pointcut that matches exactly n nested call transitions to m and o. The constructor $\mathbf{depth}^{\leq n}_{m,\underline{o}}$ defines a pointcut that matches at most n nested call transitions, and, similarly, $\mathbf{depth}^{\geq n}_{m,\underline{o}}$ at least n nested call transitions to m.

These depth constructors are useful for the definition of aspects over VPA-based protocols such as recursive protocols where advice may apply to calls at only certain (but not all) nesting levels of a protocol. For instance, in the recursive query protocol of the P2P application, let us assume that we want to optimize the underlying traversal strategy through a heuristic to perform a more superficial but faster search on nodes whose distance from the root node exceeds a certain threshold. We can easily specify such condition in the pointcut using the depth constructor $\mathbf{depth}^{\geq n}_{m,\underline{o}}$ as follows:

$$\mu a.\ depth^{\geq 5}_{query,\underline{reply}} \triangleright getCacheValue\ ;\ a$$

where the threshold is set to 5.

Permutation operators. Permutations are frequently used for the construction of protocols that allow the arbitrary interleaving of sets of events. In the presence of well-balanced contexts, interleavings of calls as well as calls and corresponding returns are subject to restrictions that cannot be modeled simply using the standard permutation function that generates

all permutations. In P2P networks, for instance, a query on one node that triggers a query on a neighbor, *e.g.,* q_1 followed by q_2, must be followed by replies in the reverse order $\underline{r_2}$, $\underline{r_1}$. The permutation $q_1, q_2, \underline{r_1}, \underline{r_2}$ is not valid. Two permutation constructors for VPA terms are provided: the $\mathbf{fp}([m, \underline{m}], [n, \underline{n}])$ constructor that defines a pointcut matching permutations of flat sequences of those pairs, *i.e.,* $m\underline{m}n\underline{n}$ and $n\underline{n}m\underline{m}$, the $\mathbf{np}([m, \underline{m}], [n, \underline{n}])$ that defines a pointcut matching permutations of nested sequences of those pairs, *i.e.,* $mn\underline{nm}$ and $nm\underline{mn}$.

Regular expressions. Regular expressions are introduced using five syntactic forms. Regular expressions can be defined as single terms ($Term$) or constructed using operators for sequencing operator ($;_{Tag}$), union ($RegEx \| RegEx$), constant times repetition ($RegEx$[CONSTANT]), and non-empty repetition ($RegEx+$).

For instance, according to the following aspect:

$$\mu a.query[5] \rhd UpdateCache\ ; a$$

caches are updated after every five queries.

4.3.2.3 Advice

An advice can be a term or a sequence of terms representing method invocations that are triggered when the corresponding pointcut successfully matches the execution trace. Note that we have deliberately restricted advice bodies to sequences of terms because we are mainly interested in aspects that are amenable to property analysis.

For instance, the advice of the following aspect consists of one instruction *createLog* that will be triggered when the *login* event is observed.

$$\mu a.login \rhd createLog\ ;\ a$$

Besides, we provide the special advice operator $\mathbf{closeOpenCalls}_{\underline{m_c}}$ that inserts a number of return transitions ($\underline{m}$) to close an open calling context. To give one example of its use, a major characteristics of a P2P network is the dynamism of peers. They can come and go unpredictably and thus it is common to have queries without corresponding replies. Subnetworks may be disconnected or messages lost. Error handling for those situations may involve the introduction of events that close a number of open recursive calls in order to skip the traversal of part of the underlying distributed network in which an error occurred. We could use VPA-based aspects to implement error handling strategies using the **closeOpenCalls** advice operator: pointcuts matching on nested contexts can then be used to restrict the application of such advice to appropriate parts of the network. The following example illustrates the use of a closing operator to add a number of "fake" replies to queries when the query exceeds a given connection timeout:

$$\mu a.\ query_f; (\underline{reply_f} \square (connectionTimeOut \rhd closeOpenCalls_{\underline{reply_f}}))\ ;\ a$$

Lastly, the advice can be defined as repetitions of a term where the number of repetitions is a constant. For instance, the following aspect will try to reconnect three times when a $connectionTimeOut$ event occurs:

$$\mu a.\ connectionTimeOut \rhd reconnect[3]\ ;\ a$$

4.3.3 Semantics

In this section we present a formal operational semantics of the VPA-based aspect language. The key idea underlying the semantics is running a visibly pushdown automaton that represents the aspect alongside the base program and try matching the execution traces of the base program with the running VPA in order to weave in the appropriate advice when a pointcut is matched. Our formalization of this idea essentially consists of the following two parts:

- The construction of a VPA for the pointcut language of VPA-based aspects, cf. Figure 4.3. This construction is formalized by a transformation from the pointcut language into definitions of VPA defined as introduced in Section 4.2.
- Define the application of VPA-based aspects, *i.e.,* how pointcuts are matched against the execution of the base program and how advice is woven into the base program execution at the corresponding join points. We have defined the weaving of VPA-based aspects by an extension of the semantics framework for regular aspects by Douence et al. [47, 48].

In the following we present the formalization of these two parts.

4.3.3.1 Overview of the VPA construction from VPA-based aspects

Let us first introduce a set of major types, variables and basic functions that we will refer to later on in the presentation of the construction of VPAs from VPA-based aspects.

- A denotes the type of aspects defined by non-terminal *Aspect* in Figure 4.3.
- P denotes the type of pointcuts defined by non-terminal *Pointcut* in Figure 4.3.
- M denotes the type of VPAs defined as introduced in Section 4.2.
- Φ denotes the table that contains environment values that are added and used through out the whole construction process. Note that transitions that handle the tag component associated to a sequence operator as well as tail recursion are only created in the last phase when the final VPA is constructed. As a consequence, it is neccessary to have a place to store the states and corresponding events that occur at these states so that these information are available later when those transitions are created for the final VPA.
- $\mathcal{T} : A \times \Phi \longrightarrow M \times \Phi$ denotes the transformation function that receives an aspect A and table Φ as inputs and returns the VPA M that defines the pointcut of aspect A while updating Φ.
- $\mathcal{R} : A \longrightarrow P$ denotes the function that removes the advice part of an aspect and returns the pointcut expression of that aspect.
- $\mathcal{V} : P \longrightarrow M$ denotes the function that constructs a VPA from a pointcut expression.
- $\mathcal{C} : M \times M \times tag \times \Phi \longrightarrow M \times \Phi$ denotes the function that concatenates two VPAs while taking into account the set tag and table Φ and returns a VPA that is the result of the concatenation while updating table Φ.
- $\mathcal{U} : M \times M \longrightarrow M$ is the function that unifies two VPAs and returns a VPA that is the result of the union.

4.3.3.2 Auxiliary functions: concatenation and union of VPAs

In the following, we first give the definition of $\mathcal{C}$ and $\mathcal{U}$ since they are basic functions that are used by the other functions.

Informally, we define the concatenation operation that connects two VPAs as follows. We add new transitions that go from states in the first VPA to initial states of the second VPA so that it is possible for the system to move from one VPA to the other one. It is possible that some events are forbidden to occur between the sequences described by these two VPAs. In such a case, we store the state and the event that is forbidden to occur from that state in a table ϕ. We then use the table ϕ and create transitions that handle forbidden events when it is the time for the construction of the final VPA. Forbidden events cause the current match to be aborted (without raising an error, which could also be raised through a simple change to our definition). Details on creating transitions handling forbidden events are presented in the end of Section 4.3.3.3.

Definition 3 (Concatenation function $\mathcal{C}$)**.** *Let:*

- $M_1 = (Q_1, Q_{in_1}, \Gamma_1, \delta_1, Q_{F_1})$ *and* $M_2 = (Q_2, Q_{in_2}, \Gamma_2, \delta_2, Q_{F_2})$ *be two input VPAs for the concatenation function.*
- $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ *be the VPA which is the result of concatenation of the two input automata with forbidden events tag, i.e.,* $M = M_1 ;_{tag} M_2$
- Φ, Φ' *respectively be the input and result tables that contains environment values of the concatenation function.*

The result VPA $(Q, Q_{in}, \Gamma, \delta, Q_F)$ *is defined as follows:*

$$
\begin{aligned}
Q &= Q_1 \cup Q_2 \\
Q_{in} &= Q_{in_1} \\
\Gamma &= \Gamma_1 \cup \Gamma_2 \\
\delta &= \delta_1 \cup \delta_2 \cup \delta' \\
Q_F &= Q_{F_2}
\end{aligned}
$$

where δ' *denotes the transition set containing transitions that are newly created. This new transition set is created as follows. Let* ts_l, ts_c, ts_r *respectively be the set of local, call, return transitions that are newly created by deviating transitions of* M_1 *which end in one of the final states of* M_1 *to all initial states of* M_2*:*

$$
\begin{aligned}
ts_l &= \{(q_0, u, q_1) \mid \exists q_0 \in Q_1, q_1 \in Q_{in_2}, q_2 \in Q_{F_1} : (q_0, u, q_2) \in \delta_1\} \\
ts_c &= \{(q_0, u, q_1, \gamma) \mid \exists q_0 \in Q_1, q_1 \in Q_{in_2}, q_2 \in Q_{F_1} : (q_0, u, q_2, \gamma) \in \delta_1\} \\
ts_r &= \{(q_0, u, \gamma, q_1) \mid \exists q_0 \in Q_1, q_1 \in Q_{in_2}, q_2 \in Q_{F_1} : (q_0, u, \gamma, q_2) \in \delta_1\}
\end{aligned}
$$

then

$$\delta' = ts_l \cup ts_c \cup ts_r$$

Let $tv = \{(q, e) \mid q \in Q_{F_1}, e \in tag\}$ *then*

$$\Phi' = \Phi \cup tv$$

That is, we add new pairs of states and events to the table Φ *so that we can mark the states and events that are disallowed to occur from those states.*

The union function is simply defined as the union of the components of the input VPAs. Our definition of this union function is exactly the same as the union operation introduced in the fundamental article on VPAs [21].

Definition 4 (Union function $\mathcal{U}$)**.** *Let* $M_1 = (Q_1, Q_{in_1}, \Gamma_1, \delta_1, Q_{F_1})$ *and* $M_2 = (Q_2, Q_{in_2}, \Gamma_2, \delta_2, Q_{F_2})$ *be two input VPAs for the union function. The union function receives two VPAs* M_1, M_2 *as inputs then returns VPA* M *defined as follows:*

$$
\begin{aligned}
Q &= Q_1 \cup Q_2 \\
Q_{in} &= Q_{in_1} \cup Q_{in_2} \\
\Gamma &= \Gamma_1 \cup \Gamma_2 \\
\delta &= \delta_1 \cup \delta_2 \\
Q_F &= Q_{F_1} \cup Q_{F_2}
\end{aligned}
$$

4.3.3.3 Mapping VPA-based aspect language constructs to VPAs

Let us recall the definition of a VPA-based aspect by non-terminal A in Figure 4.3 (for brevity, the non-terminals *Aspect, Pointcut, Advice, Tag, ID* terms in the rule definitions are abbreviated as A, P, Ad, t, a respectively):

$$A := P \mid P \triangleright Ad \mid P \triangleright Ad \;;_t\; A \mid \mu a.A \mid P \triangleright Ad \;;\; a \mid A \,\square\, A$$

At the beginning, the VPA construction algorithm will start with the given aspect A and an empty table Φ. The construction process is basically done by executing the transformation function $\mathcal{T}$ recursively starting with aspect A and table Φ as its input. At each execution of $\mathcal{T}$, the input aspect (which can be A or a part of aspect A) is checked to determine whether it can be further split into two parts or it is an atomic aspect (which cannot be decomposed). According to the definition of a VPA-based aspect that are quoted above, if the structure of the input aspect is recognized as P or $P \triangleright Ad$ (the first two rules) then we consider this aspect atomic aspect. Otherwise, if the structure of the input aspect is recognized as $P \triangleright Ad \;;_t\; A$ or $\mu a.A$ or $P \triangleright Ad \;;\; a$ or $A \,\square\, A$ (the last four rules) we consider this aspect composite aspect which can be further split into two parts. In the following we explain how we handle the aspect depending on its structure. We will start with atomic aspects first and continue with composite aspects later on.

Transformation function applied to atomic aspects. We apply the transformation function $\mathcal{T}$ to every atomic aspects to create sub-VPAs that describe the pointcut parts of these atomic aspects. As atomic aspects can be defined in only one of two formats, P or $P \triangleright Ad$, the transformation $\mathcal{T}$ is then defined as follows:

$$\mathcal{T}(A, \Phi) = (\mathcal{V} \circ \mathcal{R}(A), \Phi)$$

where the advice removing function $\mathcal{R}$ is defined as follows:

$$
\begin{aligned}
\mathcal{R}(P) &= P \\
\mathcal{R}(P \triangleright Ad) &= P
\end{aligned}
$$

That is, we first apply the removing function $\mathcal{R}$ to an aspect to obtain the pointcut. Next, we apply function $\mathcal{V}$ to the pointcut to obtain the VPA.

The pointcut transformation $\mathcal{V}$ is defined in terms of four different kinds of pointcut expressions:

$$P \quad ::= \quad Term \mid DepthOp \mid PermutationOp \mid RegEx$$

i.e., simple terms, depth-defining operators, permutation operators, and regular expression pointcuts.

The construction of one-transition VPAs for simple terms, *i.e.,* call, return, local transitions, is straightforward. Sequences of terms can also be constructed straightforwardly using the sequence transformation $\mathcal{C}$ (see definition 3).

We now consider the three remaining cases of pointcuts.

Depth-defining constructors. Depth-defining constructors basically capture events occurred at certain depths in a recursion. For a pair of call-return events, a call event increases the depth while the return one decreases it. We construct VPAs for these depth-defining constructors simply by simulating call and return events given the known depth parameter n.

Let $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ be the VPA constructed from depth constructor $\mathbf{depth}^n_{m,\underline{m}}$ where n represents the depth value and m, $\underline{m}$ respectively are the call and return events that we want to follow[2], then

$$\begin{aligned}
&Q = \{i : 0 \leq i \leq n\} \\
&Q_{in} = \{0\} \\
&\Gamma = \{\alpha\} \\
&\delta = \{(i, i+1, m, \alpha), (i+1, i, \alpha, \underline{m}) : \ 0 \leq i \leq n-1\} \\
&Q_F = \{n\}
\end{aligned}$$

where α is the stack symbol that is pushed or popped from the stack when a call m or a return $\underline{m}$ occurs.

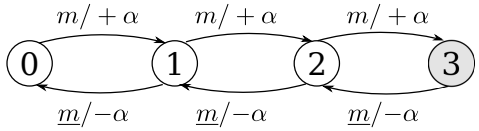


Figure 4.5: VPA constructed from depth constructor

Figure 4.5 illustrates the above VPA for $\mathbf{depth}^n_{m,\underline{m}}$ when $n = 3$. The definition of this VPA basically expresses that for every call to m there is a transition from a current state to the next one and inversely for return events. As a consequence, the accepting state (where an advice can be inserted) is the state n where the difference between the number of calls and corresponding returns equals to n. Note that by the definition of $\mathbf{depth}^n_{m,\underline{m}}$, advice is

[2]Without loss of generality, we state definitions here for call and return events of the same symbol m. The generalization to different symbols, *e.g.,* query and reply symbols, is straightforward.

only triggered by the call event but not its corresponding return. For instance, at depth level 4, a return event causes execution return to depth level 3, not triggering the advice a second time.

The VPA constructed for depth constructor $\mathbf{depth}^{\leq n}_{m,\underline{m}}$ is just similar to the above VPA, with one exception: the set of accepting states is defined as

$$Q_F = \{i : \ 1 \leq i \leq n\}$$

The VPA constructed from depth constructor $\mathbf{depth}^{\geq n}_{m,\underline{m}}$ is different from the two VPAs constructed for the depth constructors $\mathbf{depth}^{n}_{m,\underline{m}}$ and $\mathbf{depth}^{\leq n}_{m,\underline{m}}$ in that we are now interested in calls to m occurring at depth greater than n. Modeling calls and returns at depth greater than n is more tricky because we do not know in advance the maximum depth of the recursion. In order to capture these events, we construct a VPA that can differentiate the calls to m into two groups: calls at depth smaller or equal to n and calls at depth greater than n using two different stack symbols α, β. Such VPA is defined as follows:

$$\begin{aligned}
&Q = \{i\}\ \forall 0 \leq i \leq n \\
&Q_{in} = \{0\} \\
&\Gamma = \{\alpha, \beta\} \\
&\delta = \{(i, i+1, m, \alpha), (i+1, i, \alpha, \underline{m}), (n, n, m, \beta), (n, n, \beta, \underline{m}) : \ 0 \leq i \leq n-1\} \\
&Q_F = \{n\}
\end{aligned}$$

The set of accepting states contains only n, *i.e.,* a number of open calls that is at least that large. Additional call and return transitions in this state are represented by a single call and return transitions that are identified using the stack symbol β.

Figure 4.6 illustrates a VPA constructed for depth constructor $\mathbf{depth}^{\geq 3}_{m,\underline{m}}$.

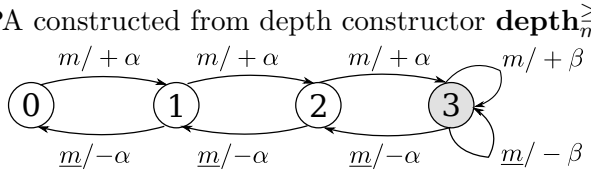


Figure 4.6: VPA constructed from a depth constructor

Permutation constructors. Basically, permutation constructors **fp, np** are unfolded into sets of sequences of individual events (terms). For example, the (nested) permutation constructor $np([e_1, \underline{e_1}], [e_2, \underline{e_2}])$ will be unfolded into $\{(e_1, e_2, \underline{e_2}, \underline{e_1}), (e_2, e_1, \underline{e_1}, \underline{e_2})\}$. A VPA will then be constructed for the corresponding sequences of events. In the following, we present how permutation constructors are unfolded.

The unfolding task includes a series of steps to turn a permutation constructor into a list/set of *valid* permutations of pairs (*i.e.,* call, return) of events. We define a list of permutations of the call events in list L_C using function `Perms`. Given a list l of events, this function is defined using (functional) list comprehensions as follows:

$$\begin{aligned}
&Perms([\emptyset]) = [[]] \\
&Perms(l) = [e \ : \ l' \mid e \leftarrow l, l' \leftarrow Perms(l \text{ - } [e])]
\end{aligned}$$

The first line of the definition states that the permutation of an empty list returns a list that includes only an empty list. The second line of the definition indicates how permutations (which are also lists of events) are generated. First, an event e of list l is chosen (expressed by the generator $e \leftarrow l$) and removed from the list l (expr. l `-` e). Next, we apply the function `Perms` over the new (sub-)list $Perms(l$ `-` $e)$. For each list l' in the list of permutations of l `-` e, the event e is appended to the head of the list l' (denoted by $e : l'$). The above procedure is repeated using the other events in list l. Since the number of events is finite, the number of permutations generated is also finite. We apply `Perms` to the list of calls to obtain the set PL_C that contains permutations of L_C. For instance, applying `Perms` to $L_C = (e_1, e_2, e_3)$ returns $\{(e_1, e_2, e_3), (e_1, e_3, e_2), (e_2, e_1, e_3), (e_2, e_3, e_1), (e_3, e_1, e_2), (e_3, e_2, e_1)\}$.

We then add the corresponding return events to the permutations to obtain valid permutations of calls and returns. These functions are defined analogous to the call permutations and can be integrated with them.

Regular expressions. Regular expressions are defined in the syntax presented in Figure 4.3 as follows:

$$
\begin{array}{lll}
RegEx & ::= & Term \\
& | & RegEx;_{Tag}\, RegEx \\
& | & RegEx[\!] \; RegEx \\
& | & RegEx \text{ [CONSTANT]} \\
& | & RegEx\ +
\end{array}
$$

We construct a VPA for a regular expression as follows. We first construct VPAs for the terms it contains and then compose the resulting VPAs to obtain the VPA for the complete regular expression.

We define the VPA representing a regular expression as follows (note that terms have already been handled above):

- In case of a sequence $RegEx ::= RegEx;_{Tag}\, RegEx$ the VPA is constructed by the concatenation (see definition 3) of two sub-VPAs representing R_1, R_2.

$$(\mathcal{V}(R_1;_{Tag}\, R_2), \Phi') = \mathcal{C}(\mathcal{V}(R_1), \mathcal{V}(R_2), Tag, \Phi)$$

- In case of an or-expression $RegEx ::= RegEx[\!] \; RegEx$ the VPA is constructed by the union (see definition 4) of two sub-VPAs representing R_1, R_2.

$$\mathcal{V}(R_1[\!]R_2) = \mathcal{U}(\mathcal{V}(R_1), \mathcal{V}(R_2))$$

- In case of a multiple transitions of the same kind $RegEx ::= RegEx$ [CONSTANT] a suitable number of VPAs is concatenated:

$$(\mathcal{V}(R[n]), \Phi') = \begin{cases} (\mathcal{V}(R), \Phi) & \text{if } n = 1 \\ \mathcal{C}(\mathcal{V}(R[n-1]), \mathcal{V}(R), \emptyset, \Phi) & \text{if } n > 1 \end{cases}$$

- In case of a repetition $RegEx ::= RegEx\ +$, we construct the VPA for R then we apply the standard Kleene plus operation on the resulting VPA as defined in Section 4.2.

$$\mathcal{V}(R+) = \mathcal{V}(R)+$$

Transformation function applied to composite aspects. How the transformation function would work on the composite aspect depends on its recognized structure.

Aspects ended by repetition variables $P \triangleright Ad \ ; \ a$ If the structure of the input aspect is $P \triangleright Ad \ ; \ a$, we split the aspect into two parts, $P \triangleright Ad$ and a. We then create the VPA M for atomic aspect $P \ \triangleright \ Ad$ and update table Φ. Formally, $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ and Φ' are defined as follows:

$$M = \mathcal{V}(\mathcal{R}(P \ \triangleright \ Ad))$$
$$\Phi' = \Phi \cup \{(q, a) \mid q \in Q_F\}$$

That is, we add pairs of final states of M and the repetition variable a to table Φ so that when we encounter μa later on we can create transitions from these final states of M to the state succeeding μa.

Sequence aspects $P \ \triangleright \ Ad \ ;_t \ A$ If the structure of the aspect is $P \triangleright Ad \ ;_t \ A$, we split the aspect into two parts, $P \triangleright Ad$ and A.

After construction of the two VPAs for $P \ \triangleright \ Ad$ and the aspect A, we construct the VPA describing the sequence $P \ \triangleright \ Ad \ ;_t \ A$ expression by concatenating (see definition 3) the VPA representing P to the VPA representing A while taking into account t. Hence, let M_1 be the VPA constructed from the $P \ \triangleright \ Ad$ expression, M_2 be the VPA constructed from A, M be the VPA constructed from the $P \ \triangleright \ Ad \ ;_t \ A$ expression, and Φ' be the updated table from Φ. Then,

$$(M, \Phi') = \mathcal{C}(M_1, M_2, t, \Phi)$$

Choice $A \ \square \ A$**.** If the structure of the aspect is $A \ \square \ A$, we split the aspect into two parts, A_1 and A_2, create VPAs for these two aspects then unify them (see definition 4). Let M_1 be the VPA constructed from aspect A_1 and M_2 the VPA constructed from aspect A_2. Then the VPA M for the choice aspect $A_1 \ \square \ A_2$ is defined as follows:

$$M = \mathcal{U}(M_1, M_2)$$

Tail-recursive aspects $\mu a.A$**.** If the structure of the aspect is $\mu a.A$, we split the aspect into two parts, μa and A. We will first create the VPA $N = (Q_N, Q_{in_N}, \Gamma_N, \delta_N, Q_{F_N})$ that represents the pointcut expression of aspect A (by applying transformation function $\mathcal{T}$ to A). Then we create the VPA $M = (Q_M, Q_{in_M}, \Gamma_M, \delta_M, Q_{F_M})$ that describes $\mu a.A$ expression as follows:

1. Define M initially as a copy of N.
2. For every transition that goes to the state that is marked with a (this information is stored in table Φ), create a new transition that starts from the same state, takes the same input, manipulates the stack in the same way but goes back to the new initial state instead.
3. Remove all transitions that end in the states that are marked with a.
4. Remove all states that are marked with a in table Φ.

In the following, we formally describe how M can be constructed from N. Let us first define several sets of state sets and transition sets involved in the construction process. Let:

- ss_1 be the state set containing all the states that are marked with a in table Φ:

$$ss_1 = \{q \mid \exists q \in Q_N : (q, a) \in \Phi\}$$

- ts_1 be the transition set containing all the transitions that end in ss_1:

$$ts = \{(q, u, q') \cup (q, u, q', \gamma) \cup (q, u, \gamma, q') \in \delta_N \mid \exists q, q' \in Q_N, \gamma \in \Gamma_N : q' \in ss_1\}$$

- ss_2 be the state set containing the starting states of all transitions in ts_1:

$$\begin{aligned} ss_2 = &\{q \in Q_N \mid \\ &(\exists q_2 \in Q_N, \gamma \in \Gamma_N : (q, u, q_2, \gamma) \in ts_1 \vee (q, u, \gamma, q_2) \in ts_1) \vee ((q, u, q_2) \in ts_1)\} \end{aligned}$$

- ts_2 be the transition set containing local transitions that are newly created by deviating all local transitions in ts_1 back to the initial state:

$$ts_2 = \{(q_0, u, q_1) \mid \exists q_0 \in ss_2, q_1 \in Q_{in_N}, q_2 \in ss_1 : (q_0, u, q_2) \in \delta_N\}$$

- ts_3 be the transition set containing call transitions that are newly created by deviating all call transitions in ts_0 back to the initial state:

$$ts_3 = \{(q_0, u, q_1, \gamma) \mid \exists q_0 \in ss_2, q_1 \in Q_{in_N}, q_2 \in ss_1, \gamma \in \Gamma_N : (q_0, u, q_2, \gamma) \in \delta_N\}$$

- ts_4 be the transition set containing return transitions that are newly created by deviating all return transitions in ts_0 back to the initial state:

$$ts_4 = \{(q_0, u, \gamma, q_1) \mid \exists q_0 \in ss_2, q_1 \in Q_{in_N}, q_2 \in ss_1, \gamma \in \Gamma_N : (q_0, u, \gamma, q_2) \in \delta_N\}$$

- ts_5 be the transition set containing the union of ts_2, ts_2, and ts_3:

$$ts_5 = ts_2 \cup ts_3 \cup ts_4$$

Then M is defined as follows:

$$\begin{aligned} Q_M &= Q_N \text{ - } ss_1 \\ Q_{in_M} &= Q_{in_N} \\ \Gamma_M &= \Gamma_N \\ \delta_M &= \delta_N \text{ - } ts_1 \cup ts_5 \\ Q_{F_M} &= Q_{in_N} \end{aligned}$$

We handle the Tag components of all sequence operators only after all sub-VPAs at lower levels are constructed recursively and composed to build the top-level VPA. We would consult the table Φ and create new transitions that go from the states marked with events in Tag components back to the initial states of the top-level VPA to obtain the final VPA that represents the pointcut of the whole aspect.

Let:

- $N = (Q_N, Q_{in_N}, \Gamma_N, \delta_N, Q_{F_N})$ be the top-level VPA that has been constructed recursively.
- ss_e be the state set containing all the states that are marked with event e in table Φ:

$$ss_e = \{q \mid \exists q \in Q_N : (q, e) \in \Phi\}$$

- te be the event set containing all events that are declared in Tag components:

$$te = \{e \mid \exists q \in Q_N : (q, e) \in \Phi\}$$

- ts be the transition set containing all the newly created transitions that go from a state in ss_e to one of the initial state of N on all events that are declared in the Tag components:

$$ts = \{(q_0, e, q_1)\} \forall e \in te, q_0 \in ss_e, q_1 \in Q_{in_N}$$

Then the final VPA $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ is defined as follows:

$$\begin{aligned} Q &= Q_N \\ Q_{in} &= Q_{in_N} \\ \Gamma &= \Gamma_N \\ \delta &= \delta_N \cup ts \\ Q_F &= Q_{F_N} \end{aligned}$$

4.3.3.4 Weaving VPA-based aspects

To define the application of VPA-based aspects to a base program, we rely on a model where aspect weaving consists of matching the pointcut of an aspect against the execution of a base program and executing the corresponding advice if the pointcut successfully matched. As defined above, a VPA is constructed for each VPA-based aspect to represent its pointcut (henceforth, we refer to this VPA as the *pointcut VPA*).

Overall, we define the matching process for a VPA-based aspect in terms of a run of its pointcut VPA in parallel with the base program. When a transition of the VPA can take place, *i.e.,* the pointcut element represented by that transition successfully matches the current join point, the corresponding aspect advice is executed. Note that, in contrast to a normal simulation of a state machine, events (*i.e.,* join points) that do not match any available transition at the current state can be simply ignored rather than causing the execution trace to be rejected by the VPA.

In the following, we formalize the weaving procedure using a small-step semantics based on the semantic framework developed by Douence et al. [47, 48], a framework originally developed for finite-state aspects that has been applied to several other aspect languages. We first present the three major components of the original framework on which the weaving semantics is based: the base program, the pointcut, and the advice. We then show how the weaving semantics of the original framework is extended to VPA-based aspects in terms of a transition relation of configurations of the base program, the pointcut VPA and VPA-based advice.

Base programs. The base program is modeled by its observable execution trace. It can be defined using a small-step transition relation $\rightarrow$ between observable states. An observable state of the base program is a configuration that includes two pieces of information: the join point and the (dynamically evolving) program state. Let (j, σ) (where j is the current join point and σ is the current program state) be the configuration of the current state and (j', σ') (where j' is the next join point and σ' is the next program state) be the configuration of the next state. The transition relation $\rightarrow$ describes how one observable state transitions to a new state:

$$(j, \sigma) \rightarrow (j', \sigma')$$

The transition relation of the base program defined in this framework is the same as the one defined in the original framework for regular aspects.

Pointcuts. As previously mentioned, a VPA that describes the pointcut is constructed for each VPA-based aspect. This VPA may evolve through its states during the execution of the base program to reflect the current state of the corresponding aspect. At each execution step of the base program, we check if the pointcut matches the current join point by checking whether its VPA can take the transition that represents the pointcut. How a VPA evolves from one state to another state is defined by another transition relation δ of a VPA.

Advice. Our advice language consists in calls to services of remote components or a *closeOpenCalls* operator. We define its semantics by adding the corresponding service calls or return actions to the trace before a join point, thus effectively defining an advice semantics corresponding to "before advice" in AspectJ-like semantics. Advice execution is modeled using an advice transition relation $\twoheadrightarrow$ that describes how the program state changes after the execution of one advice instruction. The advice transition relation is denoted as '$\twoheadrightarrow$' and defines that advice is invisible to weaving.[3] Since an advice body may include several instructions, we use $\overset{*}{\twoheadrightarrow}$ to denote the reflexive transitive closure of the $\twoheadrightarrow$ relation:

$$(\texttt{start}, \phi a_s, \sigma) \overset{*}{\twoheadrightarrow} (\texttt{end}, \phi a_s, \sigma')$$

where `start`, `end` represent the entry and exit of the advice body respectively, σ, σ' represent the current and the next program state respectively, and ϕa_s refers to the advice body to be executed. where ϕ is a substitution mapping the variables in the matching pointcut to its solutions.

The *closeOpenCalls($\underline{m_c}$)* advice operator inserts a sufficient number of returns $\underline{m}$ to close top open calls represented by a number of stack symbols c that are left on the top of the stack. If there are no c symbols on the top of the stack, nothing is done. Note that the information about the content of the stack of the VPA representing an aspect is available during aspect weaving. Hence, by inspecting the current stack content, we can determine the exact number of returns needed to be inserted by the *closeOpenCalls($\underline{m_c}$)* advice operator.

Aspect weaving. Aspect weaving of our aspect language requires the following issues to be defined:

[3]The original semantic framework for regular aspects includes an alternative transition relation for "visible advice" that is itself subject to aspect weaving. VPA-based aspects do not contribute new issues to this discussion. We have thus restricted the presentation to invisible advice.

1. Matching of elementary terms T including the handling of matching calls and returns.
2. Which aspects are applicable at a join point, taking into account depth/permutation/regular pointcuts and several aspects that may apply at once. A second problem is how to compute the follow state an applicable aspect, *i.e.,* advancing state of a VPA-based aspect.
3. How applicable aspects are woven and the program state is modified by the aspect application.

Matching of terms. At each step of the base program, matching is performed in order to check whether the current join point matches the pointcut(s) of one or more aspects.

As previously mentioned, the set of VPA-based aspects whose pointcuts participate in matching are represented by a set of corresponding VPAs. During the weaving process, these VPAs evolve to reflect the current state of their aspects. We need to know the current state and the current stack contents of a VPA in order to perform matching. Thus, at every step of the weaving process, we maintain a set V of configurations for the set of VPAs as follows:

$$V = \{(ss, \psi)\}$$

where ss is the current state set and ψ is the current stack of a VPA. There is one configuration for each VPA in the set of VPAs at any point during the weaving process.

Given the current join point j of the base execution and the set V of the current configurations of the pointcut VPAs, matching will be done for all pointcut VPAs of all aspects. The following steps are performed for matching (we henceforth use the predicate **match** j v to represent a state of a VPA matching):

1. The transitions of the pointcut VPAs which match against the current join point are determined.
2. For each local or call transition, we check wether the join point j and the input label of the VPA transition from the corresponding state of the VPA $s \in ss$ of the current configuration matches.
3. For each return transition, we have to test whether the corresponding call, that is, labeled with the right stack symbol is on top the current stack of the pointcut VPA. The information about the top of the current stack is available thanks to the ψ component of the current configuration.
4. If the current pointcut transition involves a variable, a variable assignment is returned which binds the variables used to pass information from pointcuts to advice. This assignment is empty if the match has not been successful. From a current configuration v, if there is an outgoing transition that matches the current join point j, we write **match** j $v = \phi$ where ϕ is a variable assignment.

Selection of applicable aspects and follow states. Selection is defined by the function **sel** which, given a join point j and a set of current configurations V of all pointcut VPAs, yields all applicable pointcuts (which must be single terms) and associated advice (which may be empty):

$$\textbf{sel}\ j\ V = \{(p_v, a_v)\ \mid\ v \in V,\ \textbf{match}\ j\ v\}$$

where p_v and a_v respectively denote the pointcut and advice corresponding to pointcut p_v.

As to the follow states of an aspect after weaving at a join point, note that the aspect may remain in the same configuration as before (in case none of the pointcuts associated to the current states matched the join point) or evolve to a new configuration. We define this using the function **next** which takes the current join point and the set of current configurations for all VPAs and yields the new set of configurations to be considered at the next join point:

$$\textbf{next}\ j\ V = V_1 \cup V_2$$

where

$$V_1 = \{v = (ss, \psi) \in V\} \text{ if } \textbf{not match}\ j\ v$$
$$V_2 = \{v' = (ss', \psi')\} \text{ if } \textbf{match}\ j\ v$$

where

$ss' = \{s' \mid \exists s \in ss : (s, \_, \_, s') \in \delta\}$ where $(s, \_, \_, s')$ matches any transition in δ irrespective of its type (call, return, local)

$$\psi' = \begin{cases} \psi \text{ if } (s, a, s') \text{ is a local transition, } a \text{ matches } j \\ \psi\ +\ \gamma \text{ if } (s, a, s', \gamma) \text{ is a call transition, } a \text{ matches } j \\ \psi - \gamma \text{ if } (s, a, \gamma, s') \text{ is a return transition, } a \text{ matches } j \end{cases}$$

According to the above definitions, the new set of configurations considered for the next join point is calculated by unifying two sets V_1, V_2. The set V_1 includes all the current configurations of aspects that do not evolve because their pointcuts do not match the current join point. The set V_2 includes the new configurations of aspects that have their pointcuts match the current join point. The state set ss' of a new configuration contains the states to which the matching transition moves. The content of the new stack ψ' of a new configuration is decided depending on the type of the matching transition. If the matching transition is a local one, the stack does not change. If the matching transition is a call one, we push (denoted by operator '+') γ (the stack symbol indexed by the matching transition) on the stack. Lastly, if the matching transition is a return one, we pop (denoted by operator '−') γ from the top of the stack (we do not model errors, such as non-matching calls and returns here).

Aspect weaving. We then define aspect weaving based on the base and advice transition relations $(\rightarrow, \twoheadrightarrow)$, a set of current configurations V and the advice mapping *advice*. In Figure 4.7, weaving is decomposed in two steps. First, in the weaving step proper (the topmost rule), one step of the base program applies a set of aspects and to a join point: this step selects all applicable pointcuts from the aspects, applies the corresponding advice (if possible) to the current join point, thus producing a new component state σ'. Then, the next join point that will be considered in the next round of weaving is produced. This amounts to before advice (we will discuss how after advice is handled together with an extension of advice by skipping of joinpoints in the next chapter in section 6.3.2. Finally, the aspects evolve using the function **next** and the set of new configurations, join point and base program state are returned. Second, the two bottommost aspect application rules define how the current set of applicable aspect are woven: the middle rule terminates aspect application when all aspect configurations have been woven; the bottom rule weaves one applicable aspect configuration v by determining possible variable assignments when the pointcut p_v is matched on the current

Weaving rule

$$\frac{[j,\sigma]^{\mathbf{sel}\ j\ V} \overset{*}{\longmapsto} \sigma' \qquad (j,\sigma') \rightarrow (j',\sigma'')}{(V,j,\sigma) \Longrightarrow (\mathbf{next}\ j\ V, j', \sigma'')}$$

Aspect application rules

$$[j,\sigma]^{\emptyset} \longmapsto \sigma$$

$$\frac{\mathcal{Z}' = \mathcal{Z} \backslash \{(p_s, a_s)\} \quad \mathbf{match}\ j\ v = \phi \quad (\mathbf{start}, \phi a_v, \sigma) \overset{*}{\twoheadrightarrow} (\mathbf{end}, \phi a_v, \sigma')}{[j,\sigma]^{\mathcal{Z}} \longmapsto [j,\sigma']^{\mathcal{Z}'}}$$

Figure 4.7: Weaving aspects on VPA-based protocols

join point. Finally, the rule states that the corresponding advice a_v is executed using the advice transition relation from start to end at join point j, thus producing a new protocol state σ'.

Note that, since there can be several aspects applying over the same base program, it is possible that several advice can be triggered to execute at the same time. This may cause the issue about aspect interaction where the order of advice execution may affect the results of advice application to the base program differently. Such interaction problems and how static analysis of VPA-based aspects can be used to handle it will be discussed in Section 4.5.

4.4 Visibly Pushdown Automata Library (VPAlib)

Despite the advantages of the class of visibly pushdown automata, to our knowledge, there has been no practical support available for them yet. In order to benefit from the advantages of the VPA, we have realized a library, called the `VPAlib`[97], that provides the implementation of essential data structures and operations for VPAs. This library is essential to enable the construction and verification of VPA-based aspects. The library's implementation consists of approximately 2500 lines of code, has been developed in Java SE 6, and released under LGPL license. In the following, we give an overview of the architecture of the library and then discuss the actual implementation in more detail.

4.4.1 Overview

We have striven for the development of a VPA library that could provide practical support for the implementation of the VPA-based aspect language as well as the analysis of VPA-based aspects. In principal, the library should support at least a data structure for the construction of VPAs and a set of critical methods for analysis such as intersection (*e.g.,* to check whether two VPA-based aspects might share similar pointcuts and thus apply advice at the same time) and inclusion check (*e.g.,* to check whether one VPA protocol covers and thus might substitute another VPA protocol).

Figure 4.8 shows the basic class hierarchy and some important methods and attributes of the library. The `VPAlib` library provides a set of classes to support the definition of three

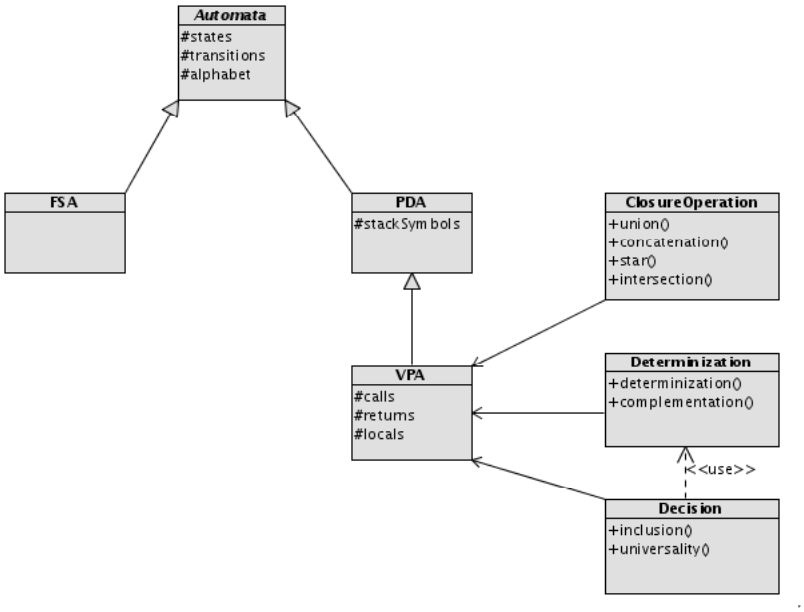


Figure 4.8: VPAlib class diagram

types of automata: Finite State Automata (FSA), Pushdown Automata (PDA), and most importantly, Visibly Pushdown Automata(VPA). Class `Automata` is the topmost one in the class hierarchy. This class features the basic components including states, transitions, and input alphabet of the simplest type of automata. Class `FSA` extends `Automata` to provide methods on finite state automata. Similarly, class `PDA` extends `Automata` to support the definition of pushdown automata. This class features the stack (and stack symbols) as an additional field.

Class `VPA` is the main class in the library since it supports the definition of visibly pushdown automata as well as the operations specific to VPAs. As previously presented in Section 4.2, a visibly pushdown automaton is different from a pushdown automaton in that its input alphabet is explicitly partitioned into three disjoint sets to represent three types of transitions: call, return, and local transitions. Therefore, class `VPA` features additional fields including *calls*, *returns*, *locals* to present such partitions of the input alphabet. In summary, every instance of class `VPA` is characterized by the set of its states, the set of initial states, the set of final states, the set of transitions, the input alphabet, three exclusive sets of three input partitions, and the set of all stack symbols.

In addition to data structures for the definition of VPAs, we aim at providing the implementation of several essential operations on VPA, including:

- Closure operations: union, concatenation, intersection for two VPAs, the complementation of a deterministic VPA, and the Kleene-* of a VPA. Note that VPA are closed under all these operations, notably intersection (an operation under which PDAs are not closed).

- Determinization: calculates the deterministic VPA from a non-deterministic VPA. Again, this operation is defined for VPAs but not for PDAs.

- Inclusion check: determines whether, given two VPAs M_1 and M_2, the language of M_1

is included in the language of M_2. This operation can be implemented thanks to the availability of the determinization operation.

- Universality check: decides whether a given VPA M accepts all strings in $\Sigma*$ where Σ represents the input alphabet of M. Similarly to the inclusion check operation, this operation can be implemented thanks to the availability of the determinization operation.

Closure operations are useful for the construction of VPAs from other primitive VPAs; the latter three operations are required for the verification of VPA protocols as well as VPA-based aspects.

4.4.2 Implementation

This section provides a brief presentation of the actual implementation of the VPA library and discusses certain issues concerning the implementation. We will first present the list of packages that have been actually implemented together with their constituents and roles in the library. We will then discuss the key ideas of the implementation of a few major VPA operations including union, intersection, concatenation, determinization, and inclusion check.

4.4.2.1 Library packages

The library contains six packages, including:

- `automata`: contains three classes `Automata`, `State`, and `Transition`. Class `Automata` is an abstract class to be extended by all the classes defining specific types of automata. Class `State` and `Transition` define respectively the basic state component and transition component of an automaton.

- `fsa`: contains class `FSA`, an inheritance of class `Automata`, that defines a finite state automaton.

- `pda`: contains three classes `PDA`, `PDATransition`, and `StackSymbol`. Class `PDA` is a subtype of `Automata` that defines a pushdown automaton. Class `PDA` features one additional important field, the set of stack symbols, implemented using class `StackSymbol`. Furthermore, class `PDATransition` is defined as an extension of class `Transition` to support PDA transitions that are capable of modifying a stack by pushing symbols onto a stack or popping symbols from a stack.

- `vpa`: this package contains the following set of eight classes for the construction of VPAs as well as the execution of VPA operations:

  - Class `VPA`, a subtype of `PDA`, defines visibly pushdown automata. It implements these automata by closely following their formal definition. That is, a VPA object has the following fields: a set of states, a set of initial states, a set of final states, a set of stack symbols, a set of transitions, and a partition of three disjoint sets of input symbols.
  - Class `VPATransition` that inherits from `PDATransition` represents VPA transitions, *i.e.,* call, return, and local transitions.

  - Class `VPAState`, an extension of class `State`, adds additional information to a state in order to support the implementation of the determinization operation for VPA and will be discussed in more details later on.
  - Class `ClosureOperation` defines several closure operations for VPAs including union, intersection, Kleene-*, complementation, and concatenation.
  - Class `Misc` defines some auxiliary methods for the implementation of closure operations.
  - Class `Determinization` defines the operation to construct a deterministic VPA from a non-deterministic one by different strategies.
  - Class `Inclusion` defines the operation to verify whether one VP language is included in another one.

- `examples`: contains a set of examples for the experimentation of different operations in the library.
- `utils`: contains a set of utility classes serving the implementation of VPA operations.

In the following subsections, we discuss the implementation of some of the principal VPA operations in more detail.

4.4.2.2 Intersection operation

The VPA intersection operation is among the set of operations implemented in the class `ClosureOperation`. Recall the fact that visibly pushdown languages are closed under intersection, which is not the case for context-free languages. Given two visibly pushdown languages L_1, L_2 that are accepted by two VPAs M_1, M_2 respectively, the intersection operation calculates the VPA that would accept the intersection of those two languages, *i.e.,* $L_1 \cap L_2$. The intersection operation can be used as a utility function in the realization of other VPA operations. For example, to decide whether one VPA includes another VPA, we have to calculate the intersection of the complement of the former one and the later then prove emptiness of that intersection. Furthermore, the intersection operation can have its own application in interaction analysis for two VPA-based protocols. This subject will be addressed later on in Section 4.5.

As previously presented in Section 4.2, given two VPAs $M_1 = (Q_1, Q_{in_1}, \Gamma_1, \delta_1, Q_{F_1})$, $M_2 = (Q_2, Q_{in_2}, \Gamma_2, \delta_2, Q_{F_2})$ with the same input partition $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ we can construct a VPA $M' = (Q_1 \times Q_2, Q_{in_1} \times Q_{in_2}, \Gamma_1 \times \Gamma_2, \delta', Q_{F_1} \times Q_{F_2})$ that represents the intersection of M_1 and M_2 where the transition relation δ' is defined as follows:

$$\text{(Local) For every } a \in \Sigma_l, \text{ if } (q_1, a, q_1'') \in \delta_1, (q_2, a, q_2'') \in \delta_2 \text{ then } ((q_1, q_2), a, (q_1'', q_2'')) \in \delta'.$$

$$\text{(Call) For every } a \in \Sigma_c, \text{ if } (q_1, a, q_1'', \gamma_1) \in \delta_1, (q_2, a, q_2'', \gamma_2) \in \delta_2 \text{ then } ((q_1, q_2), a, (\gamma_1, \gamma_2), (q_1'', q_2'')) \in \delta'.$$

$$\text{(Return) For every } a \in \Sigma_r, \text{ if } (q_1, a, \gamma_1, q_1'') \in \delta_1, (q_2, a, \gamma_2, q_2'') \in \delta_2 \text{ then } ((q_1, q_2), a, (\gamma_1, \gamma_2), (q_1'', q_2'')) \in \delta'.$$

The implementation of this intersection operation is pretty straightforward. Figure 4.9 shows part of our implementation of the intersection operation. In the beginning, we start

by calculating the product of the two initial state sets of two VPAs m_1, m_2(lines 5-7). We then explore the two VPAs m_1, m_2 in parallel from their initial states by invoking method `expand`(lines 10-12). Method `expand` (defined on lines 18-45) implements how each composite state of the resulting VPA of the intersection is further explored and how a new transition (if possible) from that state is added. From the current state, we iterate (using the for-loop in line 21) through the input alphabets to find inputs which enable transitions starting from this state on both VPAs m_1, m_2 (lines 23-24). If there is an input satisfying this condition, we add a new state which is composed by two destination states of two transitions of m_1, m_2 on that input to the resulting VPA (lines 27-32). If the two destination states are among the final states of both m_1, m_2 then the new state is a final state (lines 33-36). A new transition, created according to the above definitions, from the current state to the new state is added (lines 38-40). If the new state is not already visited, a recursive call to `expand` is made to further explore this new state (lines 42-43). The exploration terminates when there are no new states to visit.

Figure 4.10 shows an example of two VPAs M_1, M_2. Running the `intersection` method on M_1, M_2 results to the VPA illustrated by Figure 4.11. At the beginning, M_1 starts in state 0 of M_1 and M_2 starts in state 0 of M_2. The initial state of the intersection VPA is the combined state $(0, 0)$. Both VPAs can evolve on input *query*. M_1 takes the *query* transition and goes to state 1 while M_2 takes the *query* transition but stays in state 0. As a consequence, in the intersection VPA, there is a *query* transition that goes from state $(0, 0)$ to state $(1, 0)$. VPA M_1 can evolve on input *calculate* but M_2 does not accept this input so there is no corresponding transition in the intersection VPA. Finally, since both VPAs can evolve on input *reply*, the intersection VPA contains a *reply* transition from state $(1, 0)$ to state $(0, 0)$.

4.4.2.3 Concatenation operation

The concatenation operation is useful for the construction of VPAs from smaller VPAs. For instance, in the process of constructing a VPA from the pointcut of a VPA-based aspect, we usually need to construct the VPAs in the bottom-up manner where sub-VPAs are constructed first and then composed by operations such as concatenation and union to build the final VPA that describes the complete pointcut.

Figure 4.12 shows an excerpt of the implementation of the concatenation operation. The `concatenation` method calculates the VPA m as the concatenation of two VPAs m_1, m_2. The state set of m is the union of the state sets of m_1, m_2 (lines 7-8). The final state set of m is that of m_2. The initial state of m is that of m_1. The transition set of m includes two components: the union of the transition sets of m_1, m_2 (lines 16-18) and the additional transitions created to simulate the non-deterministic transition from m_1 to m_2 (lines 20-40). These additional transitions are created as follows. We iterate the transition set of m_1 (using the for-loop on line 21) and select transitions that go to the final states of m_1. For each selected transition, we create a new transition that originates from the same source state but goes to one of the initial state of m_2 instead. We add this new transition to the transition set of m.

```
public VPA intersection(VPA m1, VPA m2) {
    ...
    VPA m = new VPA();
    ...
    // Initial state
    HashSet<Pair<State, State>> pinitialstates = so.product(m1.getInitialStates(),
                    m2.getInitialStates());
    if(pinitialstates!=null){
        m.addAllInitialStates(pinitialstates);
        for (Pair<State, State> pair : pinitialstates) {
            ...
            this.expand(pair, m, m1, m2);
        }
    }
    return m;
}

private void expand(Pair<State, State> fromState,
        VPA newAutomaton, VPA m1, VPA m2) {
    ...
    for (String input : newAutomaton.getAlphabets()) {
        ...
        toState1 = m1.getStatesFromStateNInput(fromState.getFirst(), input);
        toState2 = m2.getStatesFromStateNInput(fromState.getSecond(), input);
        if (toState1.size() != 0 && toState2.size() != 0) {
            HashSet<Pair<State, State>> toStates = so.product(toState1,toState2);
            for (Pair<State, State> to : toStates) {
                ...
                if (!newAutomaton.getStates().contains(to)) {
                    existing = false;
                    newAutomaton.addState(to);
                }
                // Add to final state set
                if (m1.getFinalStates().contains(to.getFirst())
                        && m2.getFinalStates().contains(to.getSecond()))
                    newAutomaton.addFinalState(to);
                ...
                // Add new transition
                VPATransition newTransition = new VPATransition(fromState, input, to);
                newAutomaton.addTransition(newTransition);

                    if (!existing)
                        expand(to, newAutomaton, m1, m2);
    ...
}
```

Figure 4.9: Excerpt of implementation of the intersection operation on VPAs

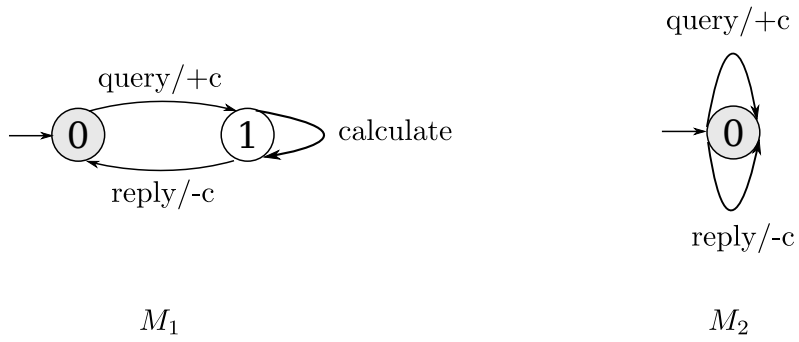


Figure 4.10: Two input VPAs M_1, M_2 for the intersection method

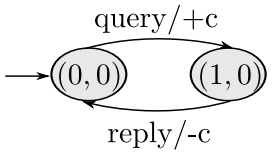


Figure 4.11: The VPA constructed as the intersection of M_1 and M_2

4.4.2.4 Kleene-star operation

The Kleene-star operation has been implemented as part of the class `ClosureOperation`. According to the formal definition of the Kleene-star operation, the VPA M* is constructed from the VPA M by a simulating M step by step and, as soon as M changes its state to a final state, M* non-deterministically updates its state to an initial state. From that point on, the stack is treated as if it were empty and states of M are tagged to distinguish old and new states.

Our implementation of the star operation is very close to its formal definition. Figure 4.13 shows part of our implementation of the Kleene star operation for VPA. First the argument VPA (line 3) is deeply copied to include the original VPA `v` in the result automaton. Then, the tagged elements are added to the original components as shown for the state set `Q` in lines 5–9. Finally, the new transition function is calculated (lines 17–55), basically by iterating over all transitions of the original transition function (for-loop in line 21) and adding corresponding new transitions as shown in the excerpt for the case of transitions of type "local" (lines 29–47).

4.4.2.5 Determinization

Unlike pushdown automata, VPAs can be determinized and thus certain problems that are undecidable for PDAs become decidable for VPAs. We have implemented the determinization operation in its own class `Determinization` in the `vpa` package to construct a deterministic VPA from a non-deterministic one. A correct implementation of this operation is a requisite for the solution of decision problems on VPAs, which are valuable for analyzing VPA protocols, such as universality and inclusion problems.

We have closely followed the definition of the determinization operation presented in Section 4.2 in our implementation. There are, however, different choices can be made about how states and transitions of the deterministic VPA are handled, notably how they are created. The "naive" approach for the implementation consists in creating the complete state set and

```
public VPA concatenation(VPA m1, VPA m2){
    ...
    VPA m = new VPA();
    // Calculating input partitions, stack alphabets
    ...
    // Union of argument state sets
    m.addAllStates(m1.getStates());
    m.addAllStates(m2.getStates());

    // Final state set
    m.addAllFinalStates(m2.getFinalStates());

    // Initial states
    m.addAllInitialStates(m1.getInitialStates());

    // Union of two transition sets
    m.addAllTransitions(m1.getTransitions());
    m.addAllTransitions(m2.getTransitions());

    // Additional transitions to non-deterministically move from M1 to M2
    for(Transition at: m1.getTransitions()){
        VPATransition t = (VPATransition)at;

        State to = t.getTo();
        State from = t.getFrom();

        if(m1.getFinalStates().contains(to)){
            VPATransition nt;
            for(State s: m2.getInitialStates()){
                if(t.getAction()==VPATransition.Action.LOCAL)
                    nt = new VPATransition(from, t.getInput(), s);
                else if (t.getAction() == VPATransition.Action.PUSH)
                    nt = new VPATransition(from, t.getInput(), s, t.getStackSymbol());
                else
                    nt = new VPATransition(from, t.getInput(), t.getStackSymbol(), s);

                m.addTransition(nt);
            }
        }
    }

    return m;
}
```

Figure 4.12: Excerpt of implementation of the concatenation operation on VPAs

```
public VPA star(VPA v) {
    // Deep copy argument to result (Q_in_res = Q_in)
    VPA res = new VPA(v);

    // Q_res = Q \cup Q^tag
    HashSet<State> taggedStates = new HashSet<State>();
    for (State s: v.getStates())
        taggedStates.add(Misc.tagState(s));
    res.addAllStates(taggedStates);

    ... // Q_F_res = Q_F \cup Q_F^tag

    ... // Gamma_res = Gamma \cup Gamma^tag

    // delta_res
    HashSet<Transition>
        delta = new HashSet<Transition>(v.getTransitions());
    VPATransition taux;
    for (Transition at: v.getTransitions()) {
        VPATransition t = (VPATransition) at;

        State q = t.getFrom();
        String a = t.getInput();
        State p = t.getTo();

        switch (t.trSort) {
        case local: {
            taux = new VPATransition(Misc.tagState(q),
                                     a, Misc.tagState(p));
            res.addTransition(taux);

            if (v.getFinalStates().contains(p)) {
                for (State r: v.getInitialStates()) {
                    taux = new VPATransition(q, a,
                                             Misc.tagState(r));
                    res.addTransition(taux);
                    taux = new VPATransition(Misc.tagState(q),
                                             a, Misc.tagState(r));
                    res.addTransition(taux); } }
             break;
        }
        case push: { ... }
        case pop: { ... }
        } // switch
    }

    return res; }
```

Figure 4.13: Excerpt of implementation of Kleene-star on VPA

then the complete transition set according to the defining rules. The problem of such an approach is the large size in terms of the number of states and transitions of the resulting VPA. This problem comes directly from the definition of the determinization operation: if the original VPA has n states, the resulting VPA will have $O(2^{n^2})$ states and $O(2^{n^2}.|\Sigma_c|)$ stack symbols, two parameters that also incur a high number of transitions. Note that a VPA resulting from the determinization operation is typically used as an input for an analysis operation, for example, inclusion check. Therefore, the size of the determinized VPA would greatly affect the feasibility of conducting analysis like that. As a consequence, it is critical to reduce the size of the resulting VPA as much as possible.

We have observed that even though the size of the state set and transition set of a determinized VPA may be formally large, many states generated according to the formal definition are not reachable from the initial state and thus can be safely excluded from the resulting VPA. As a consequence, the total number of transitions and stack symbols may be reduced significantly. Hence, our goal has been to construct a deterministic VPA while avoiding creating irrelevant states and transitions that add up to the size of the VPA but would not participate in any run of the VPA. To achieve that goal, we build the deterministic VPA "on-the-fly" while exploring the original non-deterministic VPA from its initial states instead of creating the (theoretically) complete deterministic VPA.

Figure 4.14 shows the principle part of our implementation of the determinization operation. The `determinize` method is responsible for the whole construction process to build a deterministic VPA from a non-deterministic VPA. First, the *ret* object representing the resulting VPA is created (line 2) and the initial state and input partitions are added to it. Using method `expand` we then add more states and transitions to *ret* starting from its newly created initial state q_0 (line 8). At every single state of *ret*, new states, transitions, and stack symbols are created as we explore the transitions of the original VPA. This task is performed by three methods `expandLocalTransitions`, `expandCallTransitions` and `expandReturnTransitions` (line 14-16). These three methods are also implemented using the transition relations for the deterministic VPA presented in Section 4.2.

Let us consider the body of the `expandLocalTransitions` (line 24-44) as a representative of these methods. The information we have already known is the start state $from$ which represents the start state (S, R) in the definition of the local transition relation. We have to generate the new end state (S', R') where S' and R' are defined exactly as stated in the definition (line 27-35) and then create the new corresponding transition from the start state to the end state (line 40-41). This process of generating new states and new transitions are done for all local input symbols using the `for` loop on line 26. After this process finishes, we obtain the set of the new states (represented by variable *newStates*) as a return value. Note that this set includes only new states that have not been added to the resulting VPA and thus have not been explored yet. This requirement ensures termination of the determinization algorithm since the algorithm relies on recursive calls to the `expand` method. Call transitions and return transitions are handled similarly.

After all the possible new transitions have been added to the VPA, we continue to perform the above expansion process for all the new states that have not been explored (line 18-21). This determinization algorithm terminates when all the states of the resulting VPA have been explored and there are no new state to be added.

The above implementation approach helps reducing the size of the deterministic VPA greatly comparing with the one constructed using the "naive" approach. Figure 4.15 shows a sample three-state non-deterministic VPA. This VPA has two call transitions from state 0 to

```
public static VPA determinize(VPA vpa){
    VPA ret = new VPA();
    VPAState q0 = new VPAState("q0");
    q0.addAllReachables(vpa.getInitialStates());
    q0.addAllEdges(vpa.computeIdQ());
    ret.addInitialState(q0);
    ...
    expand(q0,vpa,ret);
    return ret;
}

private static void expand(VPAState from, VPA vpa, VPA ret){
    HashSet<VPAState> newStates = new HashSet<VPAState>();
    newStates.addAll(expandLocalTransitions(from, vpa, ret));
    newStates.addAll(expandCallTransitions(from, vpa, ret));
    newStates.addAll(expandReturnTransitions(from, vpa, ret));

    for(VPAState state: newStates){
        ...
        expand(state, vpa, ret);
    }
}

private static HashSet<VPAState> expandLocalTransitions(VPAState from, VPA vpa, VPA ret){
    HashSet<VPAState> newStates = new HashSet<VPAState>();
    for(String a: vpa.getLocals()){
        for(Pair<State,State> pair: from.getEdges()){
            HashSet<State> q_primes = vpa.getStatesFromStateNInput(pair.getSecond(),a);
            for(State q_prime: q_primes){
                S_prime.add(new Pair<State,State>(pair.getFirst(),q_prime));
            }
        }
        for(State q: from.getReachables()){
            R_prime.addAll(vpa.getStatesFromStateNInput(q, a));
        }
        VPAState to = new VPAState("vpat");
        to.addAllEdges(S_prime);
        to.addAllReachables(R_prime);
        if(!ret.getStates().contains(to)) { newStates.add(to); }
        VPATransition transition = new VPATransition(from, a, to);
        ret.addTransition(transition);
    }
    return newStates;
}
```

Figure 4.14: Excerpt of implementation of determinization on VPA

state 1 and state 2 that push α to the stack, one return transition from state 1 to state 2 that pops α out of the stack, and one local transition from state 0 to state 2. State 2 is the final state and marked with a shaded circle. Figure 4.16 shows the deterministic VPA constructed from the above VPA. Theoretically, the deterministic VPA could consist of 512 (2^{3^2}) states but our implementation can produce the final VPA with only three states.

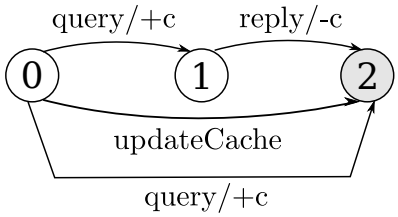


Figure 4.15: Three-state nondeterministic VPA

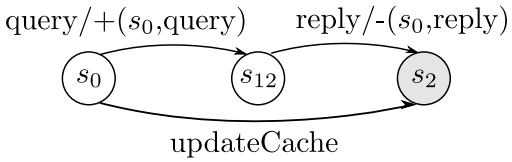


Figure 4.16: Deterministic VPA constructed from the three-state VPA

However, the current implementation of the determinization operation still suffers from the high complexity of the operation. Hence, although determinization is theoretically possible for VPAs, its complexity is still too high for the implementation to be of practical use in general. We have only verified that this operation is reasonably applicable in several practically-relevant application domains, as shown in later chapters. Defining a more efficient determinization operation for the class of visibly pushdown automata is, however, beyond the context of this thesis.

4.4.2.6 Inclusion check

Given two VPAs M_1 and M_2, the inclusion check operation confirms whether the language represented by M_1 is included by the language represented by M_2, *i.e.,* $L(M_1) \subseteq L(M_2)$. Note that this inclusion problem is undecidable for PDAs while it is decidable for VPAs. Inclusion check is especially useful for proving substitutability of VPA-based protocols as one of the pre-condition for a protocol p_1 to be substitutable by another protocol p_2 is that p_2 has to match at least the same set of traces that can be matched by p_1.

To check whether $L(M_1) \subseteq L(M_2)$, we first calculate the complement of M_2. This can be done because a VPA can be determinized and easily complemented by complementing the set of final states. Next, we take the intersection of M_1 and the complement of M_2, *i.e.,* $M_1 \cap \overline{M_2}$, then check for emptiness of that intersection. Hence, if the intersection is empty then we can conclude that $L(M_1) \subseteq L(M_2)$ and vice versa.

The inclusion check operation is implemented as a separate class in package `vpa`. Figure 4.17 shows the source code of the main method that implements the inclusion check. The `check` method checks for inclusion of two argument VPAs $M1$, $M2$. If $M2$ is nondeterministic, method `determinize`, which has already been presented above, is invoked to return the

equivalent deterministic VPA of $M2$. After that the `complementation` method returns a new VPA which is the complement of the deterministic VPA of $M2$ (lines 6-10). Next, we calculate the intersection of $M1$ and the complement of $M2$ using the `intersection` method (line 11) which has also been presented above. Finally, we check for language emptiness of the resulted VPA from the intersection calculation. The `isEmpty` method verifies whether there is any path from the initial state to a final state of a VPA. If there is no such path then this method returns *true* and vice versa.

```
    public static boolean check(VPA m1, VPA m2){
2       ...
        ClosureOperation co = new ClosureOperation();
4       VPA cm2 = new VPA();

6       if(!m2.isDeterministic()){
            cm2 = co.complementation(Determinization.determinize(m2));
8       }
        else
10          cm2 = co.complementation(m2);

12      VPA im = co.intersection(m1, cm2);

14      if(im.isEmpty())
            return true;
16
        return false;
18  }
}
```

Figure 4.17: Excerpt of implementation of inclusion check on VPA

4.5 Interaction analysis for VPA-based aspects

In this section we introduce an analysis technique that enables supports for the detection of potential interactions among VPA-based aspects. Generally, it is possible that more than one aspect is applied to a base program in parallel. In this thesis we consider the following interaction model: when pointcuts of the aspects match the same joinpoint, advice triggered by those aspects may execute at the same time and produce different results due to nondeterministic weaving.

Let us consider a simple example of interaction among the two following aspects:

$$A_1 = startDownload\ ;\ closeConnection \triangleright reconnect\ ;\ resumeDownload$$
$$A_2 = inactive\ ;\ closeConnection \triangleright removeConnection$$

where aspect A_1 will try to resume a download after the connection to the host is down while aspect A_2 will observe outgoing activities from a local node and simply remove a local connection after the connection has been remotely closed, *e.g.,* due to long period of inaction. The two aspects above may match the same joinpoint *closeConnection*, advice can thus be

triggered at the same time. Imagine that the advice of aspect A_2 is executed before A_1: an on-going download process may be lost.

The interaction analysis for VPA-based aspects we propose is based on an adaptation of the analysis of regular aspects introduced by Douence *et al.* [47, 49]. In principle, simultaneous matches of the same joinpoint or sequences of joinpoint can be statically analyzed by calculating the product automaton of two VPAs representing the pointcuts of two aspects. If the resulting automaton contains simultaneous occurrences of the same transitions in both VPAs, we conclude that there are potential interactions among two corresponding VPA-based aspects.

Note that such analysis is possible for regular and VPA-based pointcuts because the product calculation of corresponding automata decidable for both language classes. This kind of analysis is, however, infeasible for pointcuts defined using more expressive languages, such as context-free or Turing-complete ones.

The VPA library [97] described in the previous section provides a Java implementation of the product operation for two VPAs and thus can be employed for interaction analysis purpose.

In the remainder of this section we discuss Two examples of interaction analysis using the VPA library that we have performed. These examples respectively are taken from the application domains of P2P search algorithms and parallel compilation tasks. For each example, we present the corresponding pointcut VPAs and the product VPA calculated from the two aspects are shown.

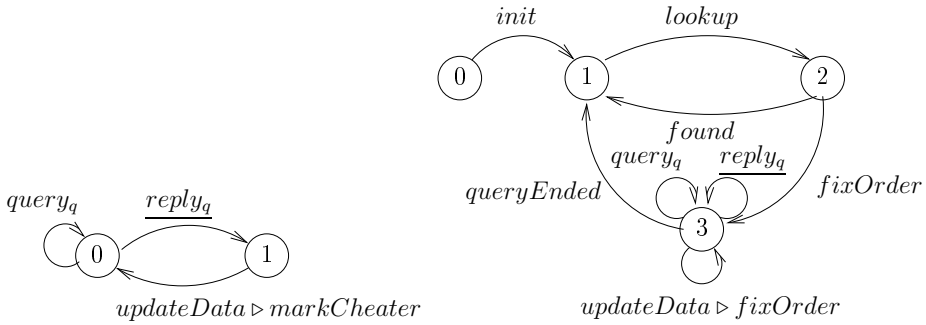
Example: P2P search algorithm. The first example uses two VPA-based aspects in the context of peer-to-peer search algorithm:

- *Trust* (see Figure 4.18(a)): This aspect is employed to query the trust information of peer nodes in the network. Starting at one node, it recursively visits all neighbors (*query*,*reply*) and updates the data structure (*updateData*) at the originating node.

- *File* (Figure 4.18(b)): This aspect works on file queries to update the preferred paths for future queries based on previous ones (a frequent optimization in P2P networks). This aspect first initializes some data structure (*init*) then attempts to look up the file locally (*lookup*) and if the file is not present locally, initiates remote queries ($fixOrder$,*query*,*reply*). Furthermore, remote queries may be ordered depending on the shared data about previous queries (*updateData*,$fixOrder$).

Since VPAs support the definition of recursive protocol, the recursive query in a peer-to-peer system can be specified naturally as call (*query*) and return (*reply*) transitions of a VPA. Furthermore, these transitions are indexed with stack symbol q so that the number of replies may not exceed the number of queries.

Note that both aspects may update the shared data structure using *updateData* and thus may interact if both are applied at the same time (which is the case in P2P networks where both types of queries are typically executed in parallel as part of maintenance operations that run in the background). Now imagine that both aspects trigger an update at the same time and that the trust query returns a result that the searched subgraph should be marked as a "cheater": in this case the update to be initiated by the file query should probably be ignored and all queries that are still on-going and that are performed within the realm of the cheater should probably be canceled.



(a) Trust query aspect (b) File query aspect

Figure 4.18: Query aspects applying over peer-to-peer search algorithm

The interaction analysis presented above is done by a calculation of the product automaton of two query aspects. Fig. 4.19 illustrates the resulting automaton. This automaton explicitly represents the interaction between both aspects on the *updateData* event. Formally, the product also indicates potential interactions on *query* and *reply*: these are, however, not interesting in the context of the example we considered, because no actions are triggered on these events. Note that events of the file query aspect which are unknown to the trust query aspect, such as *init* and *lookup*, are allowed to be interleaved in the resulting product automaton and do not affect the analysis result.

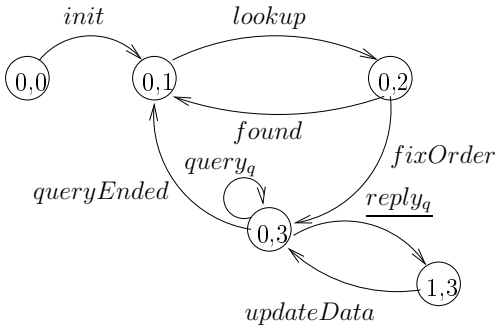
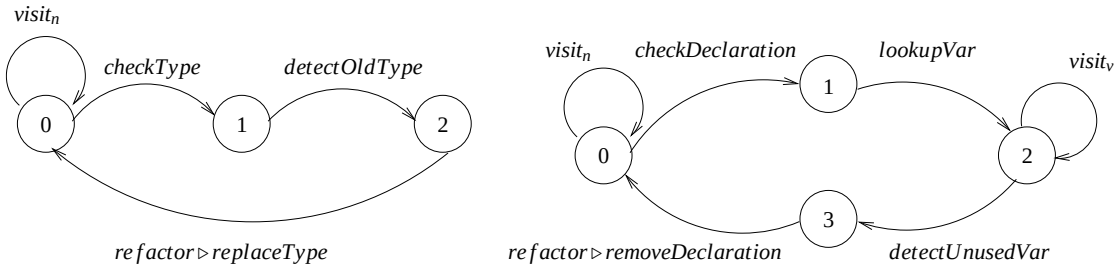


Figure 4.19: Product automaton of $Trust$ and $File$ query aspects

Example 2: compilation. This example concerns the implementation of tools for a compiler that represents programs as abstract syntax trees. Two following aspects are employed to implement two operations on the source code:

- *Replace type* (Figure 4.20(a)): This aspect recursively traverses the abstract syntax tree (using method $visit_n$) to find places where an old type is used (*checkType*). When a declaration using that specific type is found (*detectOldType*) and the source code is marked to be refactored ($refactor$), the aspect will trigger its advice to replace the old type by a new type and invoke necessary operations to ensure this modification will not cause any type mismatch error.

- *Remove declaration* (Figure 4.20(b)): Similarly, this aspect recursively ($visit_n$) looks for unused variables (*checkDeclaration*, $lookupVar$). Furthermore, another recursive traversal ($visit_v$) on the tree is needed for each variable declaration in order to conclude whether the declared variable is actually used or not. If a variable is found to be unused ($detectUnusedVar$) and the source code is marked to be refactored ($refactor$) then the aspect will trigger its advice to remove the corresponding variable declaration.



(a) Aspect replacing old types with new types (b) Aspect removing declarations of unused variables

Figure 4.20: Aspects implementing operations on an abstract syntax tree

We wish to know whether there exists any case in which the source code is marked to be refactored and the advice of two aspects both try to work on the same declaration at the same time, *i.e.,* one aspect replaces an old type with a new one while the other aspect removes the variable declaration from the source code. In the case of interaction, it is possible that an old type declaration is first replaced and the resulting declaration is then removed altogether. However, the initial modification of the type declaration is complemented by a check of consistency of type declarations at usage sites. Removal of the declaration by the second aspect will almost certainly cause errors in the compilation process.

We detect potential interactions among the two aspects above by using the intersection operation in the VPA library. Since we are only interested in simultaneous matches of join-points by two aspects, sequences of transitions of one aspect which are irrelevant to another aspect can be abstracted as one transition. The resulting product automaton illustrated in figure 4.21 shows that there are potential interactions among two aspects since both VPAs may evolve from one state to another state simultaneously on the $refactor$ event.

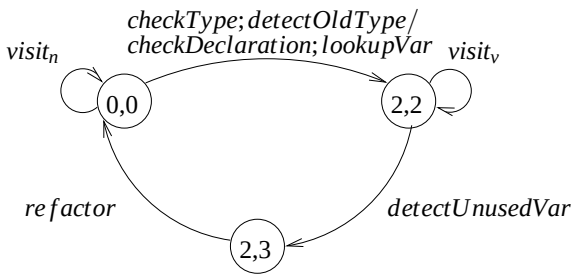


Figure 4.21: Product automaton of *Replacing_type* and *Removing_declaration* aspects

Interactions such as those in the two above examples may cause undesirable consequences to aspect-oriented systems. The detection of potential interactions by our interaction analysis

technique allows developers to reconsider the implementation of the aspects involved in order to avoid such interactions.

4.6 Conclusions

In this chapter we have introduced the VPA-based aspect language, a new AOP approach for the definition of aspects that support the class of non-regular protocols defined by visibly pushdown automata. We have shown that our VPA-based aspect language permits the declarative definition of non-regular pointcuts that involve well-balanced and/or recursive structures which cannot be described by usual regular aspects. We have introduced the syntax of the core aspect language for VPA-based pointcuts and advice. Furthermore, we have shown how this aspect language can be formally defined by describing an extension to a framework for small-step operational semantics of regular aspects. This semantics can serve as a guide for future implementation of the aspect language.

Beside the VPA-based aspect language, we have developed the VPAlib, a Java library released under LGPL license that provides an implementation of basic operations and decision procedures VPAs.

As the class of visibly pushdown languages preserves most closure and decision properties of regular languages, our VPA-based aspect language can support certain types of property analysis. Concretely, we have shown that our language supports the analysis of aspect interaction properties that rely on a notion of interaction as the simultaneous application of aspects. We have demonstrated our approach to detect interaction among VPA-based aspects through two examples. We have also used the VPAlib library to perform experiments of interaction analysis.

Chapter 5

Applications

In this chapter we present two applications of VPA-based aspects: first, apply VPA-based aspects in order to supervise access in nested login sessions in distributed systems. We study a typical setup of some remote access systems that are characteristic to, as well as (manual or automatic) nested logins using SSH [130] and Remote Desktop [11], as well as rights delegation in Enterprise JavaBeans. Often actions have to be performed that depend on the completion of potentially nested sessions that can be described concisely using VPA-based aspects.

As a second, more substantial, application in the context of grid computing system we consider an an extension of the application of VPA-based aspects to peer-to-peer protocols of the preceding chapter. In this chapter we explore further to grid systems that are deployed on peer-to-peer network. Examples of this type of system include the Jalapeno [122], Juxta-CAT [108], and OurGrid [28] systems.

In the remainder of this chapter, we describe these two types of applications in more details and motivate a set of specific cases where VPA-based aspects are useful for their definition. We do not define the corresponding base programs and protocols in terms of the existing systems, such as Jalapeno for grid computations, but abstract their functionalities and properties and define aspects over these abstract systems. There are two main reasons for the use of abstract systems instead of real ones. First, the concrete systems, such as SSH/Remote Desktop and Jalapeno/Juxta-CAT/OurGrid employ, for the problems we are considered by syntactically rather different but semantically quite similar mechanism; by using abstract systems as base programs, we can focus on more important and relevant features for the application of VPA-based aspects while skipping technical details of the implementation. As a consequence, the definitions of aspects for these systems are more clear and straightforward. Second, we can collect interesting features and properties from various similar applications and study them as part of a single abstract system.

This chapter is structured as follows. In Section 5.1 we present two examples in order to motivate the application of VPA-based aspects to the realization of typical remote access systems. In Section 5.2 we motivate the application of VPA-based aspects to peer-to-peer grid systems. We show examples of VPA-based aspects that are used to implement some functionalities of an abstraction of this kind of system. Finally, we conclude the chapter in Section 5.3.

5.1 Remote access systems

We denote as a remote access system a program and corresponding infrastructure that enable users or computational entities to log into another computer over a network, to access data and to execute commands or programs on that remote machine. Examples of this type of application include infrastructures and applications, such as SSH [130] and Remote Desktop [11]. Note that a real-world application of this kind often involves security protocols to ensure strong authentication and secure connections. Part of the secure handling of remote accesses is login session management. In the following, we show how VPA-based aspects can can be used to improve session management by making explicit certain of the corresponding protocols.

Let us assume that there is a network of N machines, each machine being identified by an address (i in the following). In order to access machine m_i, a user executes a *login* command and sends her username and password to m_i's address for verification. If the access privilege of that user to m_i is confirmed, she now can open a session and execute commands on m_i according to her privileges. After using the machine, the user is expected to close her session on that machine by executing a *logout* command. However, an inactive session after a specific period of time may be automatically closed by the administrator of the used machine for reasons of security and resource protection. Note that sessions may be nested, *e.g.,* a user can connect to a machine m_2 starting from m_1 that has been reached in turn by a preceding login from another machine m_0. In the following we discuss two advanced features for such systems, automatic session closing and that can be defined using VPA-based aspects.

Automatic session closing on behalf of users. As previously presented, a user is supposed to use the *logout* command to close a session after finishing her work on the remote machine. It is, however, quite frequent that users forget to close a remote session and just log out from the principle (local) machine. As inactive sessions may be automatically closed by the administrator, unsaved on-going work could be lost (depending on the policy regarding this matter on the remote machine). In order to avoid this potential problem, we need a functionality that automatically executes the *logout* command and extra instructions on how unsaved work should be treated on the remote machine on the user's behalf.

A simple way to realize such functionality is to add a function that will execute the *logout* command on the remote machine. This function should only be triggered if the user logs out from her local machine while the remote session is still open. If the remote session is only "one-level" from the local machine, a simple regular aspect that observes the *logout* events on the local machine then executes a *logout* command on the remote one is sufficient for the implementation of this functionality:

$$\begin{aligned} reg\_logout = \mu a.login_{local} \ ; \ \mu b.(login_{remote} \ ; \ logout_{remote} \ ; \ b \\ \square \ logout_{local} \triangleright logout_{remote} \ ; \ a \\ \square \ logout_{local} \ ; \ a) \end{aligned}$$

The above definition basically expresses that the aspect observes *login* and *logout* events on the local machine and the remote machine. If the aspect detects a logout event on the local machine without a logout event on the remote machine, it enforces the *logout* command

on the remote machine first before proceeding to the execution of the *logout* command on the local machine.

The above aspect, however, does not work with multiple-level remote sessions. For instance, in the case of two nested remote sessions, a typical sequence of *login*, *logout* events is:

$$login_0 \ ; \ login_1 \ ; \ login_2 \ ; \ logout_2 \ ; \ logout_1 \ ; \ logout_0$$

where $0, 1, 2$ indicate the login-level of the machine with respect to the local machine. When the user just execute $logout_0$ without $logout_2$ and $logout_1$ first, the logout clean-up function should be able to run these missing commands on behalf of the user. The above regular aspect cannot do this because it is only designed to work with regular sequences of events. We cannot just add more explicit session events to the pointcut definition because the depth of the remote sessions can be (theoretically) infinite.

VPA-based aspects can solve the above problem thanks to their ability to keep track of well-balanced events and the *closeOpenCall* operator. Here is the definition of the VPA-based aspect that implements the required functionality:

$$\begin{aligned} \mu a.login_{local} \ ; \ \mu b.(&login_{remote} \ ; \ b \\ &\square \ logout_{remote} \ ; \ b \\ &\square \ logout_{local} \triangleright closeOpenCall_{login_{remote}} \ ; \ a) \end{aligned}$$

According to the above definition, the aspect observes the *login* and *logout* events on the local and remote machines. *Login* events on remote machines of different levels are tracked precisely thanks to the nature of VPA-based aspects. Therefore, when the aspect detects a *logout* command on the local machine, it can generate the exact number of *logout*s corresponding to the open sessions on the remote machines in order to close those sessions on the user's behalf.

Controlling remote access privileges. A common problem of remote access is that certain machines on a network require higher level of securities than the other ones. For instance, machine m_1 can be remotely accessed from anywhere while machine m_2 only allows remote access from within two machines m_3, m_4 but not the others. In other words, if a user wants to access m_2 remotely, she has to go through m_3 or m_4. In the real world, this kind of requirement is very common, *e.g.*,, if m_2 is behind a firewall which only allows access from m_3 or m_4.

Let us consider a more specific example relating to this problem. Assume that the user is working on local machine m_0. There is a machine m_2 that she can only access if she has successfully passed through machine m_1. That means, whenever she connects to m_2 from within an open session on m_1 she should be granted access and vice versa. Some check is needed to ensure this requirement. Practical solutions for this kind of functionality exist. One possibility is to check on m_2 the address of the machine from which the connection to m_2 has been made and verify whether the address is on the allowed list. However, implementing this functionality using VPA-based aspects enables certain extra advantages.

A VPA-based aspect ensuring the above requirement, *i.e.,* access to m_2 is only granted if the connection is made from within m_1, can be defined as follows:

$$\begin{aligned}
&\mu a.login_0\ ;\ \mu b.(login_1\ ;\ \mu c.(login_2 \triangleright proceed\ ;\ logout_2\ ;\ c \\
&\qquad\qquad \square\ login_3\ ;\ \mu d.login_2 \triangleright proceed\ ;\ logout_2\ ;\ d \\
&\qquad\qquad\qquad \square\ logout_3\ ;\ c \\
&\qquad\qquad \square\ logout_1\ ;\ b) \\
&\qquad \square\ login_2 \triangleright skip \\
&\qquad \square\ logout_0\ ;\ a)
\end{aligned}$$

According to the above definition, the aspect observes *login, logout* events on machines m_0 to m_3. A user starts by connecting to the local machine m_0. She then might want to connect to machine m_1, m_2, or log off machine m_0. However, since connection to m_2 must be made from within m_1, any attempt to connect to m_2 from m_0 is rejected by the *skip* instruction[1]. On the other hand, connections to m_2 made from within m_1 or from any remote open sessions, *e.g.,* session on m_3, created from within m_1 are permitted by the *proceed* instruction (which formally is equivalent to an empty operation since our semantics uses before advice.). The *proceed* instruction acknowledges that the access is granted and thus further actions on m_2 can be taken. Note that this is not possible if we just simply check the address of the connecting machine and m_3's address is not on m_2's accepted list. It is, on the other hand, possible with VPA-based aspects. Furthermore, the implementation using VPA-based aspects is provided at protocol level and thus brings possibility for analysis on protocol compatibility.

We have motivated the application of VPA-based aspects to the realization of two functionalities of a remote access system. As we have shown in the above examples, the notion of well-balanced execution events is critical in this application context, and the pointcuts of VPA-based aspects bring concrete benefits for the functionalities that deal with nested sessions of those events.

5.2 Computational grids on peer-to-peer overlay network

Peer-to-peer networking has been proposed as an improvement over the traditional client-server Internet model for the distribution of information. Peer machines in the network can share the burden of providing resources for peers that need them without having to rely on a central server or certain specific connections. Popular file sharing systems such as Napster[10], Kazaa[9], and Gnutella[6] are examples of the peer-to-peer networking model. However, note that peer-to-peer systems and applications are not necessarily limited to file sharing systems.

Grid Computing, on the other hand, allows the combination of resources from many computers in order to perform a common task that requires large computing power. The main goal of this paradigm from a resource-usage point of view is to harness under-utilized machines and create a virtual supercomputer at a fraction of the cost of traditional supercomputers. This paradigm is nowadays used in a large number of scientific and commercial projects, *e.g.,* SETI@home[22], BEinGRID[2], European Datagrid[57].

As computational grids become more popular and attract more participants, they need flexible mechanisms in order to perform efficiently. Peer-to-peer networks with their ability

[1]The *skip* is discussed in detail in Sec. 6.3.2 on page 131, where its integration in our formal framework for VPA-based aspects is given.

to handle a large number of decentralized resources and related communications can provide a good infrastructure for computational grids. Moreover, peer-to-peer networking and grid computing share the common characteristics as distributed computing paradigms and thus are likely to be able to apply together in a system. There have been a body of works that build computational grids over peer machines in a network.

In this section, we motivate the application of VPA-based aspects to a combination of these two kinds of system. The following section is organized as follows. First, we present three existing systems of this kind of application. The architecture and basic functionalities of these systems serve as the foundation for the abstract system on which we later define aspects. We then present an abstraction of P2P-based grid computing systems that makes explicit features of the peer management functionality of the existing systems. Then we show how VPA-based aspects can be harnessed for the implementation of such features.

5.2.1 Peer-to-peer grid computing system

Generally, approaches for the combination of computational grids and peer-to-peer networking aim at creating shared grids over a peer-to-peer platform where peer nodes can submit tasks that are to be solved collectively thanks to resources contributed by many peers. The architecture of the peer-to-peer platform and the policies of task distribution and result collection vary much between such existing hybrid systems.

In the previous chapter, we have shown the ability of our VPA-based aspect language to define aspects over regular and non-regular protocols. Since P2P-based grid systems typically involve a number of communications between machines/nodes in the systems and these communications often have to comply with a set of protocols to ensure interoperability, P2P-based grid systems can be potential application targets of VPA-based aspects. Hence, we are interested in finding specific features of P2P-based grid systems that can benefit from VPA-based aspects.

In the following, we review three existing peer-to-peer grid computing systems: Jalapeno [122], Juxta-CAT [108], and OurGrid [28]. These three systems have been selected because they are sufficiently mature to identify some generally interesting features that can be represented by an easier-to-tackle abstract system. In principle, we are looking for functionalities that involve sequences of communications between different nodes in a system, *e.g.,* a function that implements the procedure in which a node submits a task to the control node to have the task processed using the grid resources.

Jalapeno is the result of a research project that aimed at developing a grid computing system based on peer-to-peer technology. The peer-to-peer platform has been realized based on JXTA technology [8]. This technology defines a set of open protocols that allows different types of devices to communicate with each other in a peer-to-peer manner. Peers in the Jalapeno network can take one or more of three roles: *manager*, *worker*, and *task submitter*. Each worker peer has to join a peer group which is then managed by a manager peer. Worker peers can communicate with other worker peers in the same group directly or to peers in other groups indirectly through their manager peers. To use a Jalapeno grid to solve a problem, a task submitter submits a collection of tasks to a randomly chosen manager peer. This manager peer then distributes a certain number of tasks to its worker peers and forwards the rest to other manager peers. This process can continue until all the tasks are assigned to worker peers. When a worker peer finishes a task, the result is returned through the manager peer to

the task submitter. Hence, in this system, tasks are supposed to be performed independently and results can be returned directly to the task submitter without having information to be propagated back on the same path through which tasks are distributed.

Juxta-CAT is another JXTA-based research platform that provides a distributed environment for job execution sharing. There are two types of peers in the Juxta-CAT platform: *brokers* and *clients*. Client peers can submit tasks to be performed by the grid. Broker peers play the administration role, *i.e.,* assigning tasks to the Grid nodes and returning results to the client peers. Hence, the broker peers play the similar role of manager peers in the Jalapeno system while client peers play the role of both task submitters and worker peers in Jalapeno. However, the Juxta-CAT system implements a more sophisticate strategy for task distribution than the Jalapeno system. For example, broker peers may consult historical and statistical data to select the best candidate peer for processing an incoming task.

OurGrid is a free-to-join peer-to-peer grid that was released in 2004. In its beginning OurGrid was also implemented upon the JXTA protocol library but now uses a selfmade peer library. The OurGrid platform is not completely decentralized because it still uses a centralized rendezvous service for peers to find each other. In contrast to Jalapeno and Juxta-CAT OurGrid is still under active development as of March 2011. Furthermore, it has a large developer as well as user community.

5.2.2 Abstract peer-to-peer grid computing system

We sketch the model of a peer-to-peer grid computing system with some functionalities and properties that have been realized by the existing systems like the above ones. Our goal is not to build another peer-to-peer grid system but to make explicit protocols that govern peer management in such systems that can benefit from VPA-based aspects. We have developed this abstraction mainly based on the Jalapeno system, a popular system whose source code could be freely accessed. However, the peers are managed similarly by the other systems.

Architecture. The computing system is constructed to work over a decentralized P2P overlay network. Peer nodes participate in the network and act both as providers and customers of the system: they can submit tasks and request for computing resources, but are also supposed to contribute a portion of their resources in order to provide computing services for other peers.

Although there will be no central infrastructure, peers are virtually organized into groups. Each group is managed by a manager peer who manages a small set of worker peers. Every individual peer has to join a group. Peers only communicate with its group members and manager.

Basically, a peer can take on one or more of three roles: worker, manager, and task submitter. A peer always starts as a worker peer when it first connects to the network. It then searches for a manager to join a group. If it cannot find a group after some time, it becomes a manager peer and creates a group and starts accepting worker peers. A peer can also submit a task to its manager and request computing service. A manager peer then splits the task into several parts and distributes them to its worker peers and other manager peers. The results will be returned to the manager peer and eventually to the task submitter.

Figure 5.1 presents the protocol of a worker peer. This protocol partially describes a session of a worker. From the initial state (state 0), the worker joins the network and the group by the *join* event. Next, the worker waits for task assignment by the manager. When the worker receives a task assignment (*getTask* event), it evolves to state 2. From this state, the worker can execute the task (state 3), finishes the task (state 4), and then returns the result to the manager (state 5). From state 2, the worker can also check whether an assigned task is removed, if so, the worker goes to state 6. The worker can also decide to quit the network and goes to state 7 without finishing the assigned task. From state 5 and state 6, the worker can quit the network or return to state 1 where it waits for another task assignment. Note that the number of *returnTask*(representing the event when the worker returns the result to its manager) and *removeTask*(representing the event when an assigned task is signaled to be removed from execution) events must match the number of *getTask* events. This matching requirement is described by the specification that defines *getTask* as call transitions and *returnTask* and *removeTask* as return transitions and the index c that is tagged to these events.

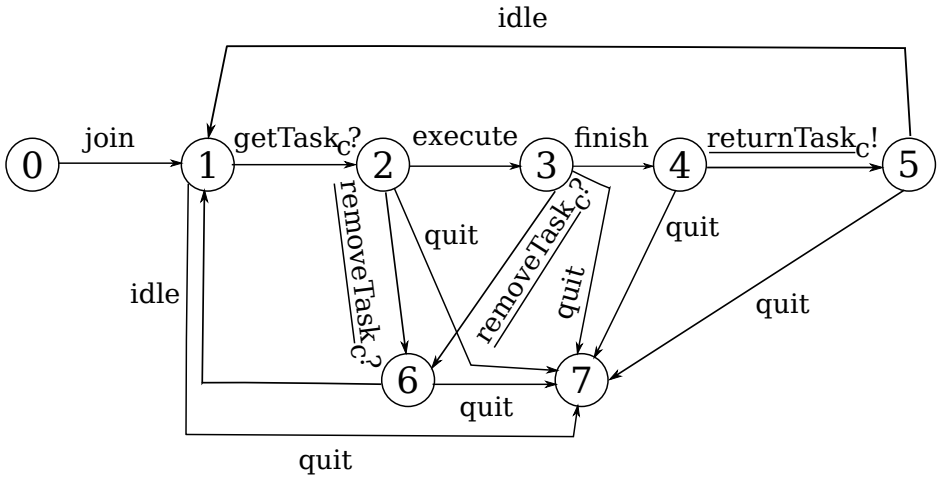


Figure 5.1: Protocol of a worker peer

Figure 5.2 presents the protocol of a task submitter. The protocol describes the transition of a task submitter in a session as follows. From the initial state (state 0), the task submitter joins the network (state 1) and prepares the task bundle (state 2). Then the task submitter looks for a manager by taking the *askManager* transition and gets back the response by taking the *replyManager* transition. These two types of transition have to match each other so we model them as call and return transitions indexed with stack symbol d. After getting information about available managers, the task submitter then selects a manager (state 3). It then sends tasks to the selected manager via *sendTask* event. It can also receive results via *getResult* or removes a task via *removeTask* event. Since the number of result messages the task submitter receives plus the number of task removing messages the task submitter sends has to match the number of tasks sent out, we specify *sendTask* as call transition and *getResult* and *removeTask* as return transitions. These transitions are indexed with symbol c. From state 3, the submitter can also quit the network (state 4) or go back for another request via *idle* event. Note that, many events in this protocol are synchronized (represented by the '?' and '!' symbols) with events in the protocol of a manager presented below.

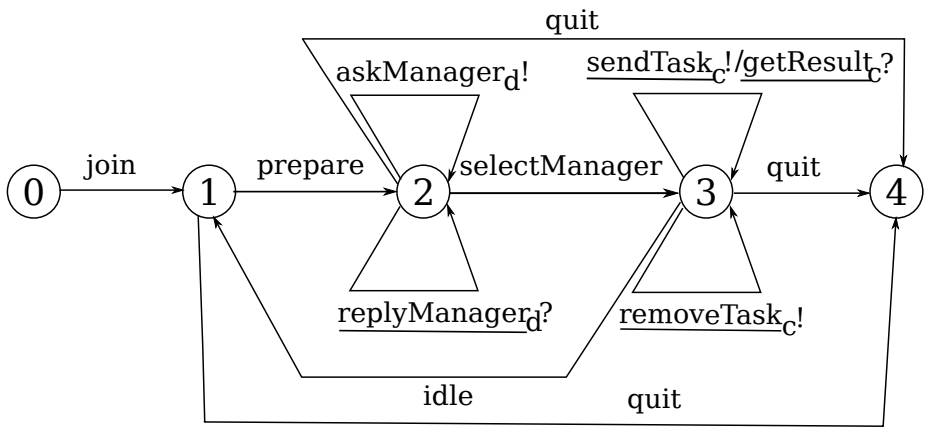


Figure 5.2: Protocol of a submitter peer

Since a manager interacts with both task submitters and workers, two protocols are defined for a manager peer. The first one (presented by Figure 5.3) describes how the manager interacts with a worker. The second one (presented by Figure 5.4) describes how the manager interacts with a task submitter. In the protocol of the manager versus a worker shown in Figure 5.3, the manager can basically send tasks to the worker (by activating *sendTask* event), get back results from the worker (*returnTask* event), or tell the worker about a removed task (*removeTask* event). Note that these three events are synchronized with events in the worker protocol. The protocol of the manager versus a task submitter shown in Figure 5.4 is similar to that of the task submitter, *i.e.,* the manager communicates with the task submitter via synchronized events including *askManager*, *replyManager*, *sendTask*, *getResult*, and *removeTask*.

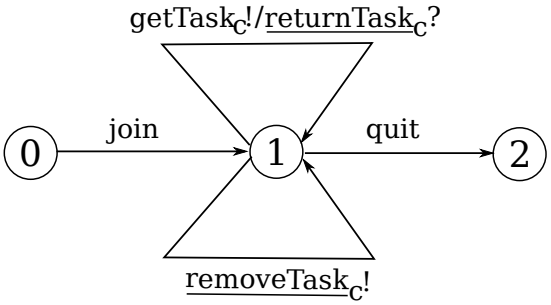


Figure 5.3: Protocol of a manager versus worker peers

5.2.3 Applications of VPA-based aspects.

In the following we demonstrate how VPA-based aspects can be used to improve three different functionalities of P2P-based grid infrastructures.

5.2.3.1 Monitoring task processing.

Assume that we need a functionality to display the status of tasks executed by workers to an interface. To implement this functionality, we basically have to observe the following events

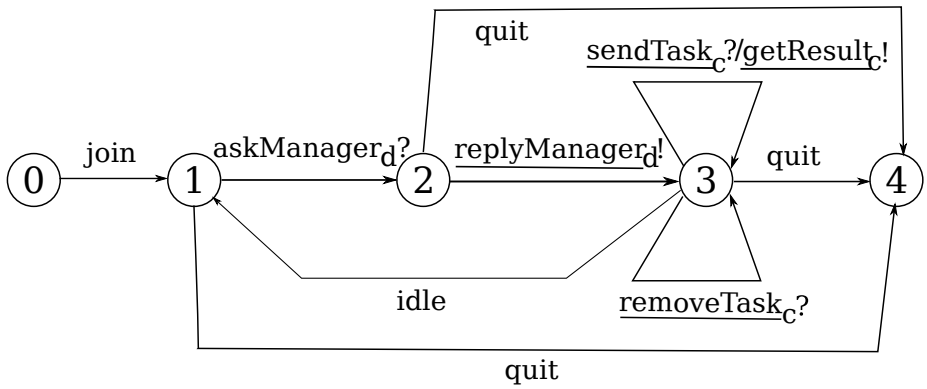


Figure 5.4: Protocol of a manager versus a submitter peer

related to a task a:

- a is sent by a task submitter (via *sendTask* event in the protocol shown in Figure 5.2).
- a is canceled by a task submitter (via *removeTask* event in the protocol shown in Figure 5.2).
- a is assigned to a worker (via *getTask* event in the protocol shown in Figure 5.1).
- a is executed by a worker (via *execute* event in the protocol shown in Figure 5.1).
- a is done and returned to a manager (via *returnTask* event in the protocol shown in Figure 5.1).

Note that these events are triggered by two different roles/components in the system. Therefore, we implement the above functionality by two aspects that match these events and updates the status of a task accordingly. VPA-based aspects that have the ability to capture pairs of matching events can be used to implement this functionality as follows (for simplicity we assume that *update* implicitly passes the matched event to the underlying task):

$$A_1 = \mu a.getTask_c \triangleright update \ ; \ execute \triangleright update \ ; \ finish \ ; \ \underline{returnTask_c} \triangleright update \ ; \ a$$
$$A_2 = \mu a.sendTask_c \triangleright update \ ; \ a$$
$$\quad \square \ \underline{getResult_c} \triangleright update \ ; \ a$$
$$\quad \square \ \underline{removeTask_c} \triangleright update \ ; \ a$$

In the above definitions, A_1 is an aspect that follows the worker protocol while A_2 is an aspect that follows the submitter protocol. Note that the *update* function is defined under the assumption that a *returnTask* event that is matched by the aspect must correspond to a *getTask* event. Therefore, in the pointcut definition of the aspect, it is necessary to specify *getTask* and *returnTask* as corresponding events. Similarly, in aspect A_2, *sendTask*, *getResult* and *removeTask* are specified as corresponding events.

Example 2: Handling task canceling. When a worker peer unexpectedly quits the network, it does not notify its manager about its status and will not return any result for tasks it has been assigned. If there is an aspect or another function that is not designed to handle the lack of results, such deviation from the basic protocol may create a problem for that aspect. For example, the above aspect that tracks processing tasks always waits for results to be returned in order to record the status of the process and update the interface. In this case, a VPA-based aspect can be used to generate messages for the system to be compatible with the aspect that needs the return message.

$$\begin{aligned}\mu a.getTask_c \;;\; execute \;;\; \underline{returnTask_c} \;;\; idle \;;\; a \\ \Box\ \underline{removeTask_c} \;;\; idle \;;\; a \\ \Box\ quit \triangleright closeOpenCall(returnTask_c) \;;\; a\end{aligned}$$

The above aspect observes events that show the status of tasks being assigned, executed, or removed so that it knows the number of on-going tasks. When an assigned task is executed and returned or has been removed, it is considered "finished". However, when a worker peer quits the network with one or more assigned tasks not yet returned or removed, the aspect generates missing *returnTask* events for those on-going tasks.

Example 3: Handling task re-distributing. Assume that we would like to apply a new rule that restricts to 20 the number of open tasks that each manager can handle. Tasks that exceed this limit are forwarded to another manager. These re-distributed tasks will be sent back to the original manager after they are finished. This new restriction rule can be implemented using the following VPA-based aspect.

$$\begin{aligned}&\mu a.depth^{\leq 20}_{getTask_c,\underline{returnTask_c}} \;;\; a \\ &\quad\Box\ depth^{\geq 20}_{getTask_c,\underline{returnTask_c}} \triangleright skip \;;\; forwardTask_c \;;\; a \\ &\quad\Box\ sendbackTask \triangleright \underline{returnTask_c} \;;\; a \\ &\quad\Box\ \underline{returnTask_c} \;;\; a\end{aligned}$$

The above aspect observes *getTask* events that occur when a manager assigns a task to a worker peer. When the manager tries to assign more than 20 tasks, the aspect stops the execution of the *getTask* event and forwards the task to another manager instead. Note that this intervention by the aspect is transparent to the manager. As a consequence, the manager is not aware of the forwarding and may not be able to handle the returned tasks. Therefore, the aspect also observes the *sendbackTask* event that occurs when a forwarded task is sent back to the original manager. It then generates a *returnTask* event for the task so that the original manager can receive the results just as a result from a task executed by its own group.

5.3 Conclusions

In this chapter we have demonstrated in the context of two application domains — remote access systems and P2P-based grids — how VPA-based aspects can be employed to improve existing or introduce new functionalities. The use of VPA-based aspects in these applications

is appropriate for a few reasons. First, thanks to its expressive pointcut language, VPA-based aspects can capture both regular and non-regular sequences of events especially well-balanced nested events. This ability is particularly useful for application domains such as P2P applications that often involve such kinds of sequences or protocols. Second, since the pointcut of a VPA-based aspect is defined by a VPA, different types of property analysis can be attempted. This possibility comes from the fact that VPAs are described by the class of visibly pushdown languages which share many features with regular languages in terms of analysis support. For instance, in Section 4.5 of the previous chapter, we have shown how analysis can be done for the detection of interaction among VPA-based aspects.

Chapter 6

Component evolution using VPA-based aspects

Component-based systems, that is, systems that are built from composition of basic building blocks, so-called software components[1], are frequently subject to continuous evolution in order to provide new or better functionality. Component-based systems often require functionalities that crosscut components, that may be defined and implemented using aspects. However, there are usually two typical limitations that hinder the application of AO languages to component-based systems. First, many aspect-oriented languages feature pointcut languages that quantify pointcuts over sets of individual join points rather than sequences of execution events. However, the latter are often very useful for crosscutting functionalities of component-based systems in order to define and manipulate interaction protocols. The ability to define evolution operations over interaction protocols should also permit that aspects can be applied in a black-box manner to components and enable the effects of aspects on component-based systems easier to control. Second, most of the current aspect-oriented languages do not support the analysis of properties of component-based systems that are subject to aspect-based evolution.

In order to address these limitations, an aspect language should provide an expressive-enough pointcut language that can capture protocols and an advice language that allow their modification. In addition to expressiveness, that aspect language should also permit the analysis of the component-based system after the application of aspects. The aspect language that we have presented in Chapter 4 is useful in this context. The VPA-based pointcut language allows us to define pointcuts that capture sequences of execution events that are governed by interaction protocols. Support for the definition of pointcuts and over protocols include all the basic regular-like operators as well as VPA-specific constructors, such as the set of depth constructors.

6.1 Example: evolving P2P systems by VPA-based aspects

In the following we exemplify the modifications of component-based systems we have in mind and the type of properties we want to be preserve in the context of the P2P-based grid

[1] In this chapter we use the notion of software components that interact via well-defined interfaces and are subject to (black-box) composition [121]

system introduced in the previous chapter. Three different types of modifications to protocols established for this system will be introduced by means of VPA-based aspects.

Extending worker protocols by acknowledgements. In the P2P-based grid system, a manager may send updates to workers in its group. Workers then get updates and run these updates locally. They communicate about these updates according to their protocols regarding this matter. More concretely, a manager uses the following protocol for sending updates:

$$p_{Manager} = start \ ; \ maintenance \ ; \ !sendUpdate_c* \ ; \ finish \ ; \ ?\underline{getAck_c}*$$

That is, a manager will start the session, do some maintenance jobs, and begins sending out updates to workers. After finishing sending all new updates to all workers, it optionally expects acknowledgement messages from its workers about the updates. Note that the number of acknowledgement messages the manager receives should not exceed the number of update messages it has sent out. Hence, the *sendUpdate* and *getAck* events are modeled as a pair of VPA call and return in VPA-based protocol language. These pairs of events are indexed by stack symbol c.

A worker peer handles updates according to the following protocol:

$$p_{Worker} = join \ ; \ ?sendUpdate_c* \ ; \ applyUpdate$$

That is, the worker peer joins the group, accepts update messages, then applies the updates. Note that this version of the worker implementation does not impose the worker send back acknowledgement messages as expected by its manager. However, two protocols are still compatible because the manager is not blocked in order to wait for acknowledgement messages and no participant in the communication sends unknown messages to the other.

Assume that we would like to upgrade the version of the worker implementation so that a worker will sends an acknowledgement message to its manager when it gets an update. As a consequence, we need to modify the protocol followed by a worker. Such a modification can be done using a VPA-based aspect defined as follows:

$$A = p_{Worker} \ ; \ eoe \triangleright closeOpenCalls(\underline{!getAck_c})$$

In the above aspect, the advice operator *closeOpenCalls* inserts a sequences of *getAck* events right after the end of the p_{Worker} protocol. The *eoe* event is a dummy event that marks the end of the event preceding it (in this case, *eoe* indicates the end of the p_{Worker} protocol). The above aspect will make a worker to send a number of acknowledgement messages equal to the number of updates the worker has received.

It turns out that the modified protocol of p_{Worker} is also compatible with the protocol $p_{Manager}$. This is because the protocol $p_{Manager}$ has been designed to accept the acknowledgements introduced by the modified protocol of p_{Worker}. We can say that compatibility is preserved for this type of aspect-based modification.

Restricting outgoing messages of managers. The second extension to the P2P-based grid system focuses on the protocols that govern the communication between a manager and a worker about task assignment. A manager sends a set of tasks to a worker according to the following protocol:

$$p_{Manager} = start \ ; \ prepare \ ; \ !getTask_d* \ ; \ ?\underline{returnTask_d}* \ ; \ end$$

That is, a manager will start the assignment session, prepare the tasks then send them to a worker. The manager may send (*getTask*) several tasks to one worker. After that, the manager waits (*returnTask*) for results from the worker and finally ends the session. Here, stack symbol d is used to specify the relation between *sendTask* and *getResult* events.

On the other hand, a worker waits and handles tasks according to the following protocol:

$$p_{Worker} = join \ ; \ ?getTask_d * \ ; \ !\underline{returnTask_d} * \ ; \ quit$$

That is, a worker will join the group, accept tasks then execute, return task results to its manager, and finally quit the group.

The above two protocols are compatible. Assume that we would like to restrict the number of tasks a manager can send to each worker to a constant k so that a worker does not have to handle too many tasks at the same time. This modification can be implemented using the following depth-cutting aspect to apply to $p_{Manager}$:

$$A = \mu a.start \ ; \ prepare \ ; \ depth^{\geq k}_{getTask_d, \underline{returnTask_d}} \rhd (skip \ ; \ saveTask) \ ; \ a$$

The above aspect is called depth-cutting aspect since it restricts the number of *getTask* events (which are actually task sending events performed by a manager) to k. After skipping a task sending event, the advice runs the *saveTask* command to save tasks that are not sent to the worker so that these tasks can be reassigned later. Hence, after the original protocol $p_{Manager}$ is modified, a manager now only sends at most k tasks to a worker. The new protocol resulted from this modification is still compatible with the original protocol p_{Worker} because p_{Worker} implies that a worker can accept any number of tasks from a manager while it will only receive at most k tasks from a manager. Hence, compatibility is preserved after the modification of the depth-cutting aspect to the system.

Extending managers by support for task bundles. Assume that managers in the P2P-based grid system are designed to satisfy the following protocol when forwarding task bundles and collecting results to and from their neighbors:

$$P = start \ ; \ selectNeighbors \ ; \ (!sendBundle_e \mid !\underline{collectResults_e}) * \ ; \ end$$

Let us refer to the above protocol as the forwarding protocol. That is, a manager first starts the session, then selects a list of neighbors that it wants to forward task bundles to and gets back results from its neighbors. In this original protocol, the *sendBundle* and *collectResults* events can occur alternatively. However, *collectResults* are considered return events of *sendBundle* and thus modeled as pairs of events indexed by symbol e. Finally a manager can end the session with the *end* event.

Assume that it is possible to have different implementations of the manager peer as long as task forwarding and collecting events generated by a manager are accepted by the originally defined forwarding protocol P. In other words, it is required that P should be able to substitute a forwarding protocol p_1 implemented for a manager peer in the system. This requirement is needed to ensure compatibility between manager peers and other roles in the system. In the first implementation of the system, the actual forwarding protocol is defined as follows:

$$p_1 = start \ ; \ selectNeighbors \ ; \ !sendBundle_e * \ ; \ !\underline{collectResults_e} * \ ; \ end$$

The above protocol p_1 is different from protocol P in that a manager will forward all task bundles first and then collect results later. This implementation of the manager satisfies the requirement regarding the forwarding protocol as sequences of events generated by p_1 are accepted by P.

Assume that we now modify the forwarding strategy of the first implementation of the system by introducing the following aspect to apply to protocol p_1:

$$A = start\ ;\ selectNeighbors\ ;\ (!sendBundle_e\ ;\ bundleSent \rhd (!sendBundle_e\ ;\ \underline{collectResults_e})*)*\ ;\ !\underline{collectResults_e}*\ ;\ end$$

When aspect A observes that a manager sends a task bundle to one of its neighbors, A inserts a sequence of pair of *sendBundle* and *collectResults* events in order to forward tasks (and collect results) to (sub-)neighbors of the original targeting neighbor. However, we still have to make sure that the modified version of p_1 by aspect A can also be substituted by the original protocol P. A closer investigation on the set of possible sequences of events generated by the modified version of p_1 confirms that this requirement is met.

In the three above cases of using aspects to introduce modifications to protocols of the P2P-based grid system, we have shown that for certain specific cases, compatibility or substitutability that is established for the original implementation can be preserved after the application of aspects. Note that we have basically relied on the structure of the protocols and aspects in order to reason about the properties for the systems in the above cases. We aim at generalizing these specific cases and defining classes of protocols and aspects where preservation of properties can be formally proved.

In the remainder of this chapter, we investigate how certain types of component evolution can be expressed using VPA-based aspects, such that fundamental correctness properties of software compositions can be guaranteed. We provide an approach that ensures the preservation of properties by construction. Instead of proving the preservation of properties for specific systems and aspects, we aim at achieving more general results by proving the preservation of properties for certain classes of aspects and systems. The property then holds for all aspects and component-based systems in the corresponding classes.

The remainder of this chapter is organized as follows. We revisit basic correctness properties for component-based systems and present our model of evolution using VPA-based aspects in Section 6.2. We then present our approach in Section 6.3. We introduce a set of theorems that allow us to establish the preservation of properties for certain classes of aspects and systems. Finally, we conclude the chapter in Section 6.4.

6.2 Component-based systems: correctness and evolution

One of the main advantages of the black-box composition of software components consists in the possibility to define the correctness of software components in terms of interactions among black-box components. The most fundamental of these correctness properties are the compatibility and substitutability properties of components. These properties ensure that components can be used to reasonably compose complex applications: compatible components can be assembled such that they correctly interact, in particular not causing deadlocks; components that are substitutable for another one may be used instead of that component without causing errors.

If component-based systems are subject to evolution, the question arises how the preservation of compatibility and substitutability can be formally guaranteed. While such guarantees are very difficult to ensure in general, components whose interactions are governed using protocols are much better tractable by analyzing the effects of evolution on the interaction protocols. In the remainder of this section we present the notions underlying our approach to the evolution of component-based systems whose evolution is defined by modifications of protocols using VPA-based aspects

In the following we introduce the compatibility and substitutability properties of software components, our model of evolution and the notion of VPA-based aspects we employ to define evolution.

6.2.1 Composition properties: compatibility and substitutability

We rely on notions of compatibility and substitutability proposed respectively by Yellin and Strom [129], and Nierstrasz [99].

6.2.1.1 Compatibility

A compatibility property of components states whether two components are compatible with each other. We have discussed in section 2.6.1 different levels of component compatibility, including compatibility at the signature level, the protocol level, and the semantics level. In our case components are equipped with explicit interaction protocols and the aspects that define the evolution of components are defined over these component protocols. These two factors make the information about protocols and modifications to protocols explicit so analysis can be performed at the protocol level.

We use the notion of protocol compatibility introduced by Yellin and Strom [129] that we present briefly in the following. Let P_1, P_2 be two collaborating protocols. A collaboration state for P_1, P_2 is a pair $\langle s, t\rangle$ where s is a state of P_1, t is a state of P_2. The set $Collabs(P_1, P_2)$ is defined to be the set of all traces that can possibly occur when P_1, P_2 collaborate. Each trace (also called collaboration history) in this trace set is a possibly infinite sequence of collaboration states starting from the initial collaboration state that comprises the initial state of P_1 and the initial state of P_2. A transition can be made from one collaboration state to another one if there is a matching event between two protocols and the direction of that event in P_1 is opposite to the direction of that event in P_2. In other words, one protocol sends the message while the other one receives it. That also means that P_1, P_2 advance synchronously on the matching message.

Two protocols have no unspecified receptions if and only if during their collaboration, when one protocol is in the state where it can send a message m, the other protocol will be in the state where it can receive that message. Two protocols are deadlock free if and only if both protocols end in their respective final states after collaborating or the collaboration can continue (in case of infinite collaboration). 'Protocols P_1 and P_2 are compatible iff they have no unspecified receptions and are deadlock free' [129]. Hence, if P_1 and P_2 are compatible, they can evolve from their initial states to their final states without any conflict. Note that this notion of compatibility allows cases where one party can receive a message yet the other party cannot send that message.

Note that in our definition of VPA-based protocol (given later in Section 6.2.3), we allow the inclusion of internal events *i.e.,* events that are not part of the collaboration between two

protocols, in a protocol declaration. Therefore, we would remove these internal events from the protocols so that we only consider events that represent a collaboration when we check for compatibility between two protocols.

Let's come back to the example system presented in the introduction of this chapter. The version of the protocol regarding task assignment in the perspective of the *Manager* component is as follows (where the '!' sign marks sent messages and '?' marks expected messages):

$$p_{Manager} = start\ ;\ maintenance\ ;\ !sendUpdate_c*\ ;\ finish\ ;\ ?\underline{getAck_c}*$$

On the other hand, a version of the corresponding protocol in the perspective of the *Worker* component is as follows:

$$p_{Worker} = join\ ;\ ?sendUpdate_c*\ ;\ applyUpdate\ ;\ !\underline{getAck_c}*$$

These two protocols are expected to collaborate on *sendUpdate* and *getAck* events. The manager can send the *sendUpdate* message to the worker then waits to receive the *getAck* message later on. On the other hand, the worker first receives the *sendUpdate* message from the manager and then sends the *getAck* message to the manager. According to the above definition of protocol compatibility, these two protocols are compatible.

6.2.1.2 Substitutability

A substitutability property of components states whether one component can be substituted by another one and preserving, at the same time, all correct interactions among the original components and other components. Similarly to the compatibility property, we base the notion of component substitutability on protocol substitutability. Note that aspects defined in the VPA-based aspect language do not change the signature of any methods implemented in a component (therefore, we do not have to worry about whether the replacing component provides at least the same services as the original component).

Substitutability of component protocol is defined using trace set inclusion: protocol p_1 is substitutable for p_2 if its trace set is a superset of the trace set generated by protocol p_2. This notion of protocol substitutability is based on the one proposed by Nierstrasz [99]. Note that in his article [99], protocol substitutability is a conjunction of two conditions: the new protocol p_1 has to generate at least the same set of sequences as p_2 and must not reject more sequences than p_2. In the context of our study, since a component only exposes the set of interaction protocols it accepts but not the ones it rejects, we have not considered the second requirement.

Let us consider two different versions of the Worker protocol:

$$p_1 = start\ ;\ selectNeighbors\ ;\ (!sendBundle_e\ |\ !\underline{collectResults_e})*\ ;\ end$$
$$p_2 = start\ ;\ selectNeighbors\ ;\ !sendBundle_e*\ ;\ !\underline{collectResults_e}*\ ;\ end$$

The above definitions show that the set of sequences of events generated by protocol p_1 include the set of sequences of events generated by p_2. For instance, both protocols accept sequence *start, selectNeighbors, sendBundle, collectResults, end* while only protocol p_2 accepts sequence *start, selectNeighbors, sendBundle, collectResults, sendBundle, collectResults, end*. Protocol p_1 is said to be substitutable for p_2.

6.2.2 Evolution model

Let us consider the basic problem defined in Section 6.2.3 in a more specific context where property Φ is a compatibility or substitutability property. Figure 6.1 illustrates our evolution model for components with interaction protocols.

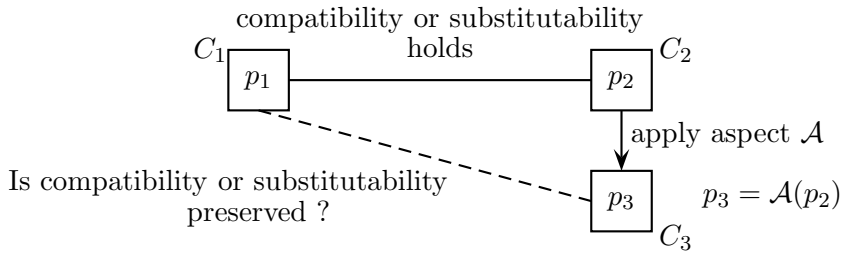


Figure 6.1: Checking for preservation of compatibility/substitutability

Starting from two protocols p_1, p_2 that govern the interactions of two collaborating components C_1, C_2 a VPA-based aspect $\mathcal{A}$ is applied to p_2 yielding the protocol p_3 that defines the interactions of the component C_3, that is C_2 after evolution. We are interested in the preservation of compatibility and substitutability properties under evolution. Given two compatible protocols p_1, p_2 we will try to prove that p_1 and p_3 are compatible. Similarly, given the fact that p_2 is substitutable for p_1 (or vice versa), we will try to prove that p_3 is also substitutable for p_1 (or p_2 in the inverse case).

6.2.3 VPA-based aspects for evolution

The evolution problem we are interested in tackling can therefore be defined as follows:

1. Given a component with a protocol p in a component-based system that satisfies property Φ.

2. Apply VPA-based aspect A whose pointcut is defined over protocol p to modify the protocol and thus the component.

3. Prove that the new system with the modified component also satisfies Φ.

In the above definition, a component is represented by its interfaces which generally expose its services and a part of its behaviors through one or more interaction protocols. A protocol p of interest is one of the interaction protocol followed by the component. The protocol is specified in our VPA-based protocol specification language given by Figure 6.2.

This protocol specification language is derived from the pointcut language of the VPA-based aspect language presented in Chapter 4, and admits not only regular sequences of events but also VPA-like sequences of events. This language essentially includes a subset of language constructs of the pointcut language that defines VPA-based aspects. Furthermore, we add extensions for component interactions, notably terms for the explicit expression of outgoing and incoming events, using the symbols '!','?'

The '!' symbol preceding a term in a protocol of a component indicates that the term represents an action invoked by this component. Conversely, the '?' symbol preceding a term in a protocol of a component indicates that the term represents an action that this component expects to happen or an incoming event that this component accepts.

$$
\begin{array}{lll}
P & ::= & P \ ; \ P \\
 & & \mid \ Term \ \mid \ DepthOp \\
Term & ::= & ?Term \ \mid \ !Term \ \mid \ \mathrm{ID} \ \mid \ \mathrm{ID}_{\mathrm{ID}} \ \mid \ \underline{\mathrm{ID}_{\mathrm{ID}}} \\
DepthOp & ::= & \mathbf{depth}^{\mathrm{CONSTANT}}_{\mathrm{ID}_{\mathrm{ID}},\underline{\mathrm{ID}_{\mathrm{ID}}}} \ \mid \ \mathbf{depth}^{\leq \mathrm{CONSTANT}}_{\mathrm{ID}_{\mathrm{ID}},\underline{\mathrm{ID}_{\mathrm{ID}}}} \ \mid \ \mathbf{depth}^{\geq \mathrm{CONSTANT}}_{\mathrm{ID}_{\mathrm{ID}},\underline{\mathrm{ID}_{\mathrm{ID}}}}
\end{array}
$$

Figure 6.2: Syntax of VPA-based protocol language

6.3 Proving property preservation by exploiting characteristics of classes of protocols and aspects

The proof or test of properties of systems that are subject to modifications by aspects are typically performed by analyzing the woven program resulting from the integration of the aspect into the original system. However, this approach has a few important limitations.

- The woven system frequently is large, often too large to be tractable for (semi-)automatic proof-support systems.
- Proofs based on the woven system are specific to aspects and original base system. Any change to the aspects or the base system require to prove the property anew.
- This approach often requires complete information, in particular information of the implementation of software components, not an option in our case since we follow a black-box approach to composition.

In general, the preservation of a property Φ after an aspect-based evolution of a component-based system S essentially depends on two factors: the evolution aspect and the interface of the components which are modified by the aspects. In our case, VPA-based aspects are used to modify components whose interface is defined through VPA-based component protocols. Because of their limited expressiveness, VPA-based aspects and protocols allow some important properties to be proven simply by considering properties of the aspect language only. We therefore propose to exploit the characteristics of visibly pushdown languages in order to reason about the aspects and the protocols. Instead of analyzing the woven system, we aim at establishing the property for the system just based on the knowledge of the aspect and the protocols that define an evolution operation.

In order to overcome the three limitations discussed above, we have developed an approach that supports correct evolution by *construction*. Concretely, we provide pattern-based aspect definitions that can be proven correct for certain sets of VPA-based (base) protocols.We thus prove properties for classes of aspects that are applied to classes of protocols depending on the constructs of the VPA-based aspect language .

This approach is not subject to the three limitations discussed above: The correctness proofs do not involve or only involve abstract properties of the woven program; the proofs support properties over classes of aspects and base programs; and the properties we consider

only involve aspects and protocols, no information on the implementation of protocols is involved.

Although this approach will not be able to achieve completeness in the sense that all possible properties can be proved for all classes of protocols and aspects, it is valuable in that it provides a non-trivial set of classes of protocols and aspects that are useful in P2P-based or even more general application systems.

We have realized our approach by first classifying aspects and protocols into specific classes based on their definitions. Basically, aspects are classified by the structure of their pointcuts and advice. Protocols are classified based on how specific kinds of events such as pairs of opening and closing actions are used. For instance, protocols that involve nested pairs of actions $m - \underline{m}$ can be classified into one class. Aspect advice can modify the protocol by inserting actions and/or removing (skipping) actions. Therefore, they are classified based on the specific type of modifications that they implement. For example, aspects that employ the specific operator *closeOpenCalls* to generate closing actions are grouped into one class. Our goal then is to seek for each combination of a class of aspect and a class of protocol a set of properties that are preserved by aspect-based evolution.

We have proved the preservation of compatibility and substitutability for a number of classes of aspects and protocols using our approach that are presented in the following. In the following subsections we respectively present three classes of aspects and their properties that we have chosen since they cover many important applications of nested structures expressible using VPA-based aspects:

- Correction of unbalanced nested call structures.
- Manipulation of the depth of nesting structures.
- Insertion of matching nested call and return events.

6.3.1 Aspects with *closeOpenCalls* advice

In this section we study the class of aspects that use *closeOpenCalls* advice and their effects on the preservation of two kinds of properties mentioned above. The $closeOpenCalls(\underline{m_c})$ advice featured by the VPA-based aspect language inserts a number of VPA return transitions $\underline{m_c}$ that are neccesary to close all open calls indexed by stack symbol c that are left on the top of the stack.

This advice can be used to implement error handling strategies by generating missing returns. It can also be used to introduce extensions to a protocol by adding sequences of return events if they do not exist in the original protocol.

In the following we present a set of properties that are preserved by aspect-based evolution using *closeOpenCalls* advice. For each type of properties, we first define a set of protocol and aspect classes that are involved. We then formally define and prove substitutability and compatibility properties of these classes.

Let $\mathcal{P}^m_{Opening}$ be the following class of protocols:

$$p = p' \ ; \ m_c * \ ; \ p''$$

where both sub-protocols p', p'' must not include any call or return event related to m.

The above definition expresses that p is a composition of three parts: sub-protocol p', sequences of calls to m tagged by symbol c, and sub-protocol p''. p represents a (typical)

class of protocols, we consider other protocol classes below, that contain open calls and may be subject to evolution.

Let $\mathcal{A}^m_{Closing}$ be the class of aspects that employ the *closeOpenCalls* advice to close open calls in a protocol p defined as follows:

$$A = p \ ; \ eoe \triangleright closeOpenCalls(\underline{m_c});$$

that is, the aspect A adds a sequence of returns $\underline{m_c}$ to the end of p. We remind the reader once again that *eoe* event is a dummy event that marks the end of the event preceding it. The presence of *eoe* event in the aspect is necessary so that events executed by aspect advice will be inserted after the end of p (but before *eoe* because our aspect language features before-advice). The number of $\underline{m_c}$ in the sequence equals to the number of c symbols (which have been pushed into the stack by corresponding calls of $\underline{m}$) that are on the top of the stack. We now investigate the composition properties of protocols that are defined by applying this aspect class to classes of protocols.

Substitutability. Substitutability is preserved after the application of aspect of class $\mathcal{A}^m_{Closing}$ if certain conditions on the target protocol and its counterpart are satisfied.

Theorem 3 (Substitutability)**.** *Let:*

- p_1, p_2 *be protocols of class* $\mathcal{P}^m_{Opening}$ *and* p_1 *can be substituted by* p_2*.*
- A *be an aspect of class* $\mathcal{A}^m_{Closing}$*.*
- $p_3 = A(p_2)$*, i.e.,* p_3 *is the protocol resulting from the application of* A *to* p_2*.*

Then, p_1 *can also be substituted by* p_3

Proof. Let

$$p_2 = p_2' \ ; \ m_c* \ ; \ p_2''$$

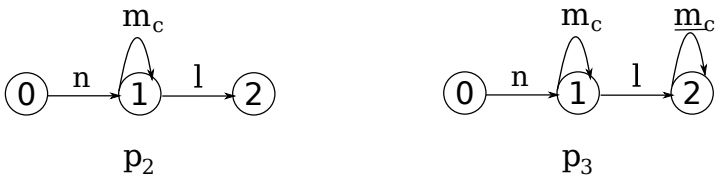


Figure 6.3: Abstract VPAs representing p_2 and p_3

We first construct the abstract VPA shown in Figure 6.3 that can represent p_2 which is a protocol of class $\mathcal{P}^m_{Opening}$. This VPA abstracts from the events not involving call and return events involving m. (We do not have to consider the protocols parts not involving m_c because p_2 is substitutable for p_1.) This VPA has three basic states $0, 1$, and 2. State 0 is the initial state of the protocol. This is the starting state of sub-protocol p_2'. The whole sub-protocol p_2' is abstracted by event n and represented by a transition from state 0 to state 1. State 1 is the state where calls m_c take place. In the definition of the protocol, m_c* represents a sequence of calls m_c so m_c events are modeled by transitions looping at state 1. Next, the

transition from state 1 to state 2 represents the sub-protocol p_2''. The whole sub-protocol p_2'' is abstracted by event l and represented by a transition from state 1 to state 2.

As we assume that p_1 can be substituted by p_2, the set $L(p_1)$ of possible events generated by p_1 must be included by the set generated by p_2, *i.e.,* $L(p_1) \subseteq L(p_2)$.

When aspect A applies to p_2, it adds a sequence of returns $\underline{m_c}$ to p_2 at then end of p_2, *i.e.,* state 2, and results in the new protocol p_3 (as shown in Figure 6.3). Hence, the set $L(p_2) \subseteq L(p_3)$ as $L(p_3)$ includes all the sequences of events generated by p_2 plus the additional sequence of returns $\underline{m_c}$.

Since $L(p_1) \subseteq L(p_2)$ and $L(p_2) \subseteq L(p_3)$, $L(p_1) \subseteq L(p_3)$, *i.e.,* the set of possible sequences of events generated by p_3 also includes the set generated by p_1. Therefore, we conclude that p_1 is also substituted by p_3. Hence, substitutability is preserved. □

Compatibility. Compatibility is also preserved for certain protocol and aspect classes involving *closeOpenCalls* advice. Table 6.1 presents several definitions of classes of protocols and aspects that we consider. We define six protocol classes (whose names start with $\mathcal{P}$) and two aspect classes (names starting with $\mathcal{A}$). Since the direction of message exchanges is essential to compatibility between components we distinguish in the following sending and receiving actions by prefix symbols '!' and '?' respectively. (We haven't done so for substitutability because substitutable protocols should be executed by the same partner, thus preserving sends or receives.)

Table 6.1: Definitions of certain protocol and aspect classes

Class Name	Definition
$\mathcal{P}^m_{io}$	$p = p' \ ; \ ?m_c* \ ; \ p''$
$\mathcal{P}^m_{oo}$	$p = p' \ ; \ !m_c* \ ; \ p''$
$\mathcal{P}^m_{oo-ic}$	$p = p' \ ; \ !m_c* \ ; \ p'' \ ; \ ?\underline{m_c}*$
$\mathcal{P}^m_{io-ic}$	$p = p' \ ; \ ?m_c* \ ; \ p'' \ ; \ ?\underline{m_c}*$
$\mathcal{A}^m_{ic}$	$A(p) = p \ ; \ eoe \ \triangleright \ ?closeOpenCalls(\underline{m_c}) = p \ ; \ eoe \ \triangleright \ ?\underline{m_c}*$
$\mathcal{A}^m_{oc}$	$A(p) = p \ ; \ eoe \ \triangleright \ !closeOpenCalls(\underline{m_c}) = p \ ; \ eoe \ \triangleright \ !\underline{m_c}*$

Protocol class $\mathcal{P}^m_{io}$ includes protocols that contain a sequence of repeated incoming call events m. The other protocol classes similarly contain sequences (possibly alternating ones) of repeated incoming and outgoing events. Aspect class $\mathcal{A}^m_{ic}$ inserts a sequence of repeated incoming return events m to the end of a protocol p. Finally, aspect class $\mathcal{A}^m_{oc}$ inserts a sequence of repeated outgoing return events m to the end of a protocol p.

Note that we use the compatibility condition as defined by Yellin and Strom [129]. This notion of compatibility allows cases where one party can receive a message yet the other party cannot send that message.

Theorem 4 (Compatibility-1)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{oo}$, p_2 *be a protocol of class* $\mathcal{P}^m_{io}$
- p_1, p_2 *are compatible*
- A *be an aspect of class* $\mathcal{A}^m_{ic}$.
- $p_3 = A(p_2)$, *i.e.,* p_3 *be the modified version of* p_2 *resulting from applying* A *to* p_2.

Then, p_1 and p_3 are also compatible.

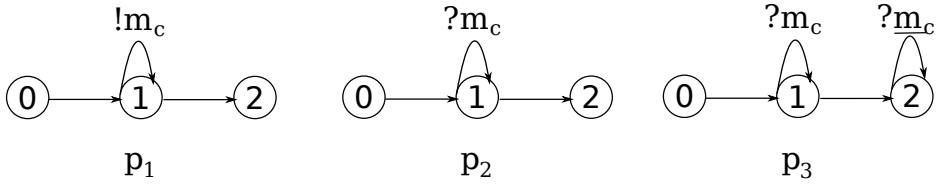


Figure 6.4: Abstract VPAs representing p_1, p_2, and p_3

Proof. Figure 6.4 shows the abstract VPAs representing p_1, p_2 and p_3. As we assume that p_1, p_2 are compatible, they both can evolve from their initial states (state 0) to their final states (state 2) without any conflict. Note that in state 1, protocol p_1 may send m_c while p_2 can always receive m_c. (If p_2 cannot accept m_c when p_1 send such event, two protocols would not be compatible.)

When aspect A is applied to p_2, a sequence of incoming returns $\underline{m_c}$ is added to the end of p_2 (state 2) to result to p_3. Hence,

$$p_3 = p_2 \ ; \ eoe \ \triangleright \ ?\underline{m_c}* = p_2' \ ; \ ?m_c * \ ; \ p_2'' \ ; \ ?\underline{m_c}*$$

Two protocols p_1, p_3 should be already compatible from state 0 to state 2 because p_1, p_2 are compatible and p_3 differs from p_2 only from state 2. In state 2, p_3 can always receive outgoing returns $\underline{m_c}$. However, since p_1 never sends such events, the additional part of the protocol does not change compatibility that has been established for p_1, p_2. Therefore p_3 and p_1 are also compatible. □

In the following we consider protocols that can be suitably extended by closing calls.

Theorem 5 (Compatibility-2)**.** *Let:*

- *p_1 be a protocol of class $\mathcal{P}^m_{oo-ic}$, p_2 be a protocol of class $\mathcal{P}^m_{io}$*
- *p_1, p_2 are compatible*
- *A be an aspect of class $\mathcal{A}^m_{oc}$.*
- *$p_3 = A(p_2)$, i.e., p_3 be the modified version of p_2 resulting from applying A to p_2*

Then, p_1 and p_3 are also compatible.

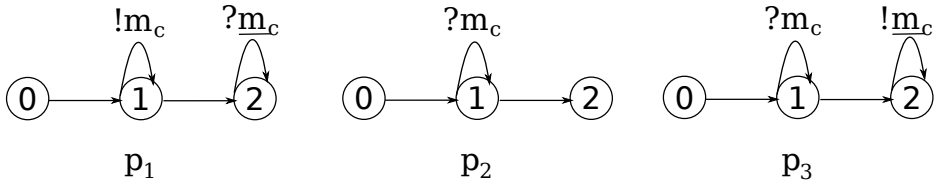


Figure 6.5: Abstract VPAs representing p_1, p_2, and p_3

Proof. Figure 6.5 shows the abstract VPAs representing p_1, p_2 and p_3. As we assume that p_1, p_2 are compatible, they both can evolve from their initial states (state 0) to their final

states (state 2) without any conflict. When aspect A is applied to p_2, a sequence of outgoing returns $\underline{m_c}$ is added to the end of p_2 to result to p_3. Hence,

$$p_3 = p_2 \ ; \ eoe \ \triangleright \ !\underline{m_c}* = p_2 \ ; \ ?m_c* \ ; \ p_2 \ ; \ !\underline{m_c}*$$

The protocols p_1, p_3 are compatible from state 0 to state 2 because p_1, p_2 are compatible and p_3 differs from p_2 only by its behavior in state 2. In state 2, p_3 may send returns $\underline{m_c}$. Since p_1 accepts a sequence of incoming returns $\underline{m_c}$ in state 2, it can accept the additional sequences added by the aspect. Therefore p_3 and p_1 are also compatible. □

Theorem 6 (Compatibility-3)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{oo}$, p_2 *be a protocol of class* $\mathcal{P}^m_{io-ic}$
- p_1, p_2 *are compatible*
- A *be an aspect of class* $\mathcal{A}^m_{oc}$.
- $p_3 = A(p_1)$, *i.e.,* p_3 *be the modified version of* p_1 *resulting from applying* A *to* p_1

Then, p_2 *and* p_3 *are also compatible.*

Proof. The proof is the dual proof (that changes sending actions into receiving ones and vice versa) of the preceding one.

□

6.3.2 Depth-dependent aspects

Recursive distributed algorithms frequently depend on actions in specific contexts, notably at specific depths of nested calls. A common example are heuristics that are formulated in terms of the traversal depth from the node where the search has been initiated, e.g., in order to determine notions of locality of computation. Since VPA-based aspect allow the explicit definition of aspects in terms of the nesting depth using the pointcut operator $depth^{\geq k}_{\underline{m_c},m_c}$, corresponding compositional properties can be proven in terms of properties of this operator and classes of protocols to which it is applied.

Table 6.2 presents the definitions of four protocol classes and the aspect class relating to depth-dependent functionality.

Table 6.2: Definitions of certain protocol and aspect classes

Class Name	Definition
$\mathcal{P}^m_{io}$	$p = p' \ ; \ ?m_c* \ ; \ p''$
$\mathcal{P}^m_{oo}$	$p = p' \ ; \ !m_c* \ ; \ p''$
$\mathcal{P}^m_{io_k}$	$p = p' \ ; \ ?m_c[k] \ ; \ p''$
$\mathcal{P}^m_{oo_k}$	$p = p' \ ; \ !m_c[k] \ ; \ p''$
$\mathcal{A}^m_{Depthcut_k}$	$A = depth^k_{m_c,\underline{m_c}} \triangleright skip$

Protocol class $\mathcal{P}^m_{io}$ includes protocols that contain a sequence of repeated incoming call events m. Protocol class $\mathcal{P}^m_{oo}$ includes protocols that contain a sequence of repeated outgoing call events m. Protocol class $\mathcal{P}^m_{io_k}$ includes protocols that contain a sequence of k repeated

incoming call events m. Protocol class $\mathcal{P}^m_{oo_k}$ includes protocols that contain a sequence of k repeated outgoing call events m. Aspect class $\mathcal{A}^m_{Depthcut_k}$ includes aspects that skip all the call events m that are occur at depth k or greater.

Note that we introduce here the *skip* command to be used in the advice of an aspect. Unlike other commands used in the advice, this *skip* command will not insert anything but rather prevent the matching event to occur. The weaving process presented in Chapter 4 describes the behaviors of "before advice", because the weaving rules first weave advice and then execute the matched joinpoint. Hence, that weaving process is not suitable for advice with a *skip* command since no execution of the matching join point should take place. Therefore, the original weaving process described in Figure 4.7 must be extended to handle *skip* advice.

The *skip* command could be implemented using a preprocessing step as follows. We have to annotate the base program in order to insert specific events that mark the point *after* the matched join point (we call these events after-events in the following). In the pointcut definition of the aspect, we replace the joinpoint that should be skipped by the after-event). Hence, only aspects that include *skip* advice will match an after-event join point. After this preprocessing step, weaving is performed as usual. In particular, the only straightforward change in the weaving definition (Fig. 4.7 on page 90) is that the new joinpoint j' is the after-event of the current joinpoint in the case of *skip*-advice.

Theorem 7 (Substitutability-1)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{io_k}$, p_2 *be a protocol of class* $\mathcal{P}^m_{io}$*; or alternatively be* p_1 *a protocol of class* $\mathcal{P}^m_{oo_k}$ *and* p_2 *a protocol of class* $\mathcal{P}^m_{oo}$
- A *be an aspect of class* $\mathcal{A}^m_{Depthcut_k}$
- p_3 *be the result of applying* A *to* p_2*, i.e.,* A *skips call events* m *from depth* k

If p_1 *can be substituted by* p_2 *then* p_1 *can also be substituted by* p_3*.*

Proof. Both p_1, p_2 send a sequence of calls m_c but p_1 only sends a maximum number of k m_c. Therefore, the set of possible sequences of events sent by p_1 is included by the set of possible sequences of events sent by p_2, *i.e.,* $L(p_1) \subseteq L(p_2)$. Hence, p_1 can be substituted by p_2. When aspect A is applied to p_2, it restricts the maximum number of calls m_c that p_2 can send to k. After application of the aspect p_2 belongs to the protocol class $\mathcal{P}^m_{io_k}$ which is the same as that of p_1. Therefore p_3 can be substituted by p_1. □

Theorem 8 (Compatibility-1)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{io}$, p_2 *be a protocol of class* $\mathcal{P}^m_{oo}$
- p_1, p_2 *are compatible*
- A *be an aspect of class* $\mathcal{A}^m_{Depthcut_k}$
- p_3 *be the result of the following application of* A *to* p_2*, i.e.,* A *skips outgoing call events* m *from depth* k

Then, p_1 *and* p_3 *are compatible.*

Proof. When aspect A is applied to p_2, aspect A restricts the maximum number of calls m_c that p_2 can send. Hence, the order of events specified by the new protocol p_3 is the same as the order of events specified by the original protocol p_2. Only the number of possible m_c events is restricted. Since p_1 does not impose any constraint on the number of calls m_c it can receive, it can collaborate with the new protocol. Therefore p_3 and p_1 are also compatible. □

Theorem 9 (Compatibility-2)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{oo_k}$, p_2 *be a protocol of class* $\mathcal{P}^m_{io}$
- p_1, p_2 *are compatible*
- A *be an aspect of class* $\mathcal{A}^m_{Depthcut_k}$
- p_3 *denote the application of* A *to* p_2*, i.e.,* A *skips incoming call events* m *from depth* k

Then, p_1 *and* p_3 *are also compatible.*

Proof. p_1 and p_2 are compatible, in particular, because p_1 can send a sequence of maximum k calls m_c while p_2 can receive a sequence of calls m_c of any length. When aspect A is applied to p_2, aspect A restricts the maximum number of calls m_c that p_2 can receive to k. Since p_1 only sends at maximum k calls m_c, the new protocol p_3 can accept the calls m_c of p_1. Therefore p_3 and p_1 are also compatible. □

6.3.3 Aspects inserting pairs of events

The aspect classes in this group represent aspects that add pairs of call and corresponding return events to a protocol. In table 6.3 we define these aspect classes and the protocol classes which are their targets of modification.

Table 6.3: Definitions of certain protocol and aspect classes

Class Name	Definition
$\mathcal{P}^m_{oa}$	$p = p' \ ; \ (!m_c \ ; \ !\underline{m_c}) * \ ; \ p''$
$\mathcal{P}^m_{oc}$	$p = p' \ ; \ (!m_c \mid !\underline{m_c}) * \ ; \ p''$
$\mathcal{P}^m_{os}$	$p = p' \ ; \ !m_c * \ ; \ !\underline{m_c} * \ ; \ p''$
$\mathcal{P}^m_{cc}$	$p = p' \ ; \ (?m_c \mid ?\underline{m_c}) * \ ; \ p''$
$\mathcal{A}^m_{oc}$	$A = p' \ ; \ !m_c \ ; \ eoe \triangleright (!m_c \mid !\underline{m_c})*$
$\mathcal{A}^m_{os}$	$A = p' \ ; \ !m_c \ ; \ eoe \triangleright \ !m_c * \ ; \ !\underline{m_c}*$
$\mathcal{A}^m_{oa}$	$A = p' \ ; \ !m_c \ ; \ eoe \triangleright \ (!m_c \ ; \ !\underline{m_c})*$

In the following we present some fundamental properties of these aspect classes and sketch proofs for those properties.

Theorem 10 (Substitutability-1)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{oc}$
- p_2 *be a protocol of one of three classes:* $\mathcal{P}^m_{oa}$, $\mathcal{P}^m_{oc}$, $\mathcal{P}^m_{os}$
- A *be an aspect of one of three classes:* $\mathcal{A}^m_{oa}$, $\mathcal{A}^m_{oc}$, $\mathcal{A}^m_{os}$

- p_3 *be the result of the application of* A *to* p_2

If p_2 *can be substituted by* p_1 *then* p_3 *can also be substituted by* p_1*.*

In the following we present proofs for one case of the theorem where p_2 is a protocol of class $\mathcal{P}^m_{os}$ and A is an aspect of class $\mathcal{A}^m_{oa}$. Other cases of the theorem can be proved using similar reasoning.

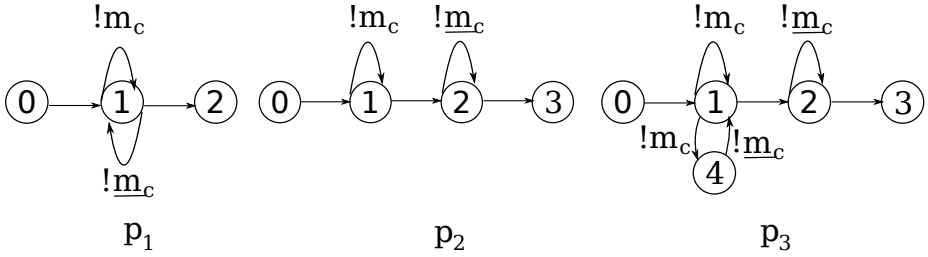


Figure 6.6: Abstract VPAs representing p_1, p_2, and p_3

Proof. Figure 6.6 illustrates the abstract VPAs representing three protocols p_1, p_2, p_3. Protocol p_1 sends three different kinds of sequences: a sequence that consists of only calls m, a sequence that consist of only returns $\underline{m}$, and a sequence of both calls m and returns $\underline{m}$. Protocol p_2 sends a sequence of m events followed by $\underline{m}$ events. Hence, since p_1 can cover all the execution of p_2, *i.e.,* the set $L(p_2) \subseteq L(p_1)$, p_2 can be substituted by p_1.

When aspect A of class $\mathcal{A}^m_{oa}$ applies to p_2, it adds a sequence of repeated sequences of $!m_c$; $!\underline{m_c}$ to every occurence of $!m_c$ at state 1 (which is the only state where m_c occurs) (see Figure 6.6).

The new protocol p_3 is defined as follows:

$$p_3 = p' \ ; \ [!m_c \mid (!m_c \ ; \ !\underline{m_c})] * \ ; \ !m_c * \ ; \ p''$$

Note that, p_1 is defined as follows:

$$p_1 = p' \ ; \ (!m_c \mid !\underline{m_c}) * \ ; \ p''$$

That is, p_1 can generate all the sequences generated by p_3. Therefore, p_3 can also be substituted by p_1. In other words, substitutability is preserved. □

Theorem 11 (Compatibility-1)**.** *Let:*

- p_1 *be a protocol of class* $\mathcal{P}^m_{cc}$
- p_2 *be a protocol of class* $\mathcal{P}^m_{oa}$ *or* $\mathcal{P}^m_{os}$
- p_1*,* p_2 *are compatible*
- A *be an aspect of class* $\mathcal{A}^m_{oc}$
- p_3 *be the result of the application of* A *to* p_2

Then, p_1 *and* p_3 *are also compatible.*

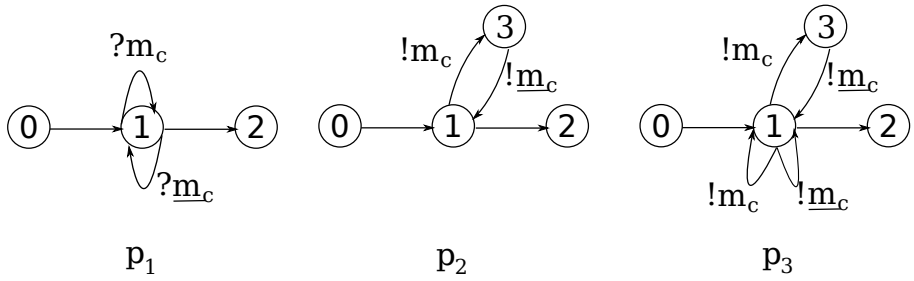


Figure 6.7: Abstract VPAs representing p_1, p_2, and p_3

In the following we present proofs for one case of the theorem where p_2 is a protocol of class $\mathcal{P}^m_{oa}$. The other case of the theorem where p_2 is a protocol of class $\mathcal{P}^m_{os}$ can be proved using similar reasoning.

Proof. Figure 6.7 illustrates the abstract VPAs representing three protocols p_1, p_2, p_3. Protocol p_1 basically receives a sequence of calls m and/or returns $\underline{m}$ while p_2 receives a sequence of pairs $m, \underline{m}$. Two protocols p_1, p_2 are compatible because p_1 can always accept sequences of $m, \underline{m}$ sent by p_1. Note that, while p_2 only sends sequences of flat pairs of $m, \underline{m}$, p_1 can accept three different kinds of sequences: a sequence that consists of only calls m, a sequence that consist of only returns $\underline{m}$, and a sequence of both calls m and returns $\underline{m}$ (where the number of calls and returns do not have to be equal). When aspect A of class $\mathcal{A}^m_{oc}$ applies to p_2 to produce new protocol p_3, it allows p_3 to send sequences that consist of any number of m and/or $\underline{m}$ (but the number of $\underline{m}$ events cannot exceed the number of m events). Since these kinds of sequences are always accepted by p_1, p_1 and p_3 are also compatible. Hence, compatibility is preserved for these particular protocol and aspect classes. □

6.4 Conclusions

In this chapter we have considered problems of aspect-based evolution on components based on VPA-based interaction protocols. Concretely, we have discussed the following issues. First, in order to be applicable in a component-based system, an aspect language should be expressive enough to be able to capture interaction protocols of components and to define modifications to those component protocols. Furthermore, the ability for aspects to manipulate interaction protocols keeps the application of aspects to be less invasive to components. Second, there is a need for a method to analyze the properties of component-based systems that are subject to aspect-based evolution. This kind of analysis helps ensuring that the modifications introduced by an aspect do not introduce problems to a component system.

We have presented a concrete approach to address these two problems. Our approach harnesses the VPA-based pointcut language that permits the specification of interaction protocols in terms of sequences of events, and provides a set of advice operators that allow us to define modifications to component protocols. We have shown how our aspect language enables the analysis of properties of compositions that are subject to evolution. To this end we have proposed several syntax-defined classes of protocols and evolutions that cover three fundamental means of evolution that are specific to VPA-based protocols. Concretely, we have studied the evolution properties of three specific VPA-based advice operations: (i) closing of open call events, (ii) constraining the depth of nested interacting structures, and (iii)

the insertion of nested pairs of interactions.

After a motivation of the corresponding evolution scenarios in the context of the evolution of P2P systems, we have shown how to prove the preservation of three types of (classes of) composition properties in the presence of evolution: compatibility and substitutability properties among interacting components, as well as more specific VPA-based composition properties.

Our approach is attractive notably for two reasons. First, the reasoning procedure for the preservation of property of a system is rather simple once the effects of the aspect on the system has been formally proved. Second, since property preservation is proved for protocol classes instead of individual protocols, the proven properties are applicable to a large number of concrete evolution scenarios.

However, our approach has its own limitations. First, our approach is incomplete: we have chosen classes of protocols and aspects for evolution that cover the most fundamental evolution scenarios that are specific to VPA-based protocols. Other scenarios exist that are not covered by the classes discussed here. Furthermore, there are obviously other composition properties that are of interest in evolution scenarios. Once again, we have only covered the most fundamental ones, notably compatibility and substitution that form the cornerstone of any theory of component composition. Second, not all property preservation can be established just by exploiting the characteristics of the aspect language. There are evolution scenarios where we need to know every detail of the actual system in order to reason about the effects of aspect-based evolution, that is, the scenarios cannot be abstracted into those covered by general classes of protocols and aspects. Finally, the method to establish "ready-to-use" properties of aspect operators for protocol classes requires significant expertise.

Chapter 7

Model checking VPA-based aspect-oriented programs

7.1 Introduction

The ability of aspects to modify a program's execution makes it more difficult to predict the behavior of the woven system. The constructive approach to the definition of correctly evolving component based systems that has been presented in the previous chapter, proposes a partial answer to this challenge. In this chapter we investigate verification techniques based on model checking [20, 40] as a complementary means to ensure the correctness of the composition of aspects and the base program.

In recent years, there have been a number of studies, *e.g.,* [77, 73, 76, 45, 116, 59], on the use of model checking technique to verify aspect-oriented systems. Generally, this technique relies on using a model checker to perform verification systematically on the model of a system against a property in order to conclude whether the property is satisfied by the system. In Section 3.3.1, we have reviewed two important model checking approaches for aspect-oriented programs. The first approach [59] proposes to build the input model from the aspect and the assumptions of the aspect about the base program and then to use the model checker NuSMV [38] for the verification. This approach allows modular verification in the sense that the checking process is performed on the aspect and the assumptions of the aspect about the base program but not the actual base program itself. The second approach [76] is based on model checking technique using CTL [39]. According to this approach, the states of the system model are first labeled with (sub-formulas of) the property. The labels on the states around the application of the aspect advice are then consulted in order to reason about the property. Both approaches employ fairly complicated algorithms that are only applicable to aspects that have regular-like pointcuts to construct verification models. Hence, it is difficult to use these approaches on VPA-based aspects. As a consequence, we have considered these approaches references and developed our own approach for model checking VPA-based aspect programs.

The remainder of this chapter is organized as follows. In section 7.2 we propose our approach to apply model checking on VPA-based models. We then present the framework we have developed for our approach in Section 7.3. In section 7.4, we compare several existing model checkers with respect to different characteristics that have determined our choice of a concrete model checker to be used for the experiments we have conducted. In Section 7.5, we

perform experimental verifications on some examples with a model checker and then evaluate the results. Section 7.6 concludes the chapter.

7.2 Motivation and approach

In order to verify a system using model checking, one needs to provide two essential inputs: a system model, which is typically defined as a transition system, and a property to be verified, which is often expressed as a temporal logic formula. A model checker then performs verification of the property on the system. When a VPA-based aspect is woven into a base program, we expect to obtain a VPA-based system. As we wish to use model checking on VPA-based systems, we would need a model checker that is capable of verifying visibly pushdown models.

However, currently there is no model checker for VPA-based systems. Existing model checkers are only applicable to systems that can be described by (some variants of) finite state automata (which are less expressive than VPAs). All model checking approaches that have been proposed for aspect-oriented programs also only deal with finite state system model. Furthermore, the inclusion check operation (which is essential for building a model checker) on VPAs is still too expensive for an implementation of a model checker to be practical. As a consequence, we cannot use existing model checkers, notably solutions such as those introduced in Section 3.3.1, directly to VPA-based systems.

Hence, instead of model checking a VPA-based system model, we propose to use a more abstract model, a finite-state based system, that is derived from the VPA-based model and then run an existing model checker on this abstract model. There is a trade-off with this approach in that we have to sacrifice some accuracy of checking the exact system model. Our goal is to obtain an abstract model which approximate closely VPA-based properties so that verification results on the abstract model provides us with sufficiently precise results about the VPA-based system model.

7.3 A framework for model checking VPA-based AO programs

In this section, we present the framework that we have developed for verifying VPA-based aspect-oriented programs by using model checking. We first formally define the framework in section 7.3.1. We then present our method for constructing an abstract system model from a VPA-based model in Section 7.3.2.

7.3.1 Model checking framework

A model checking process for an AO system needs three different inputs: a *base program* S, an *aspect* A and a *property* P. The goal is to check whether the augmented program built from a base program S and an aspect A satisfies a property P. The base program is supposed to be described by a VPA. The aspect is (an abstraction of) a VPA-based aspect. The property can be a LTL or a CTL formula. The form of the property definition is decided later based on which specific model checkers are used to check the abstract model.

Our model checking procedure is illustrated in figure 7.1. In the figure, rectangle boxes represent input or output data, and the diamond-shaped ones represent processing steps. The procedure starts by producing an abstract transition model from the VPA-based augmented

system model and then uses a specific model checker to verify this abstract transition model against the given property.

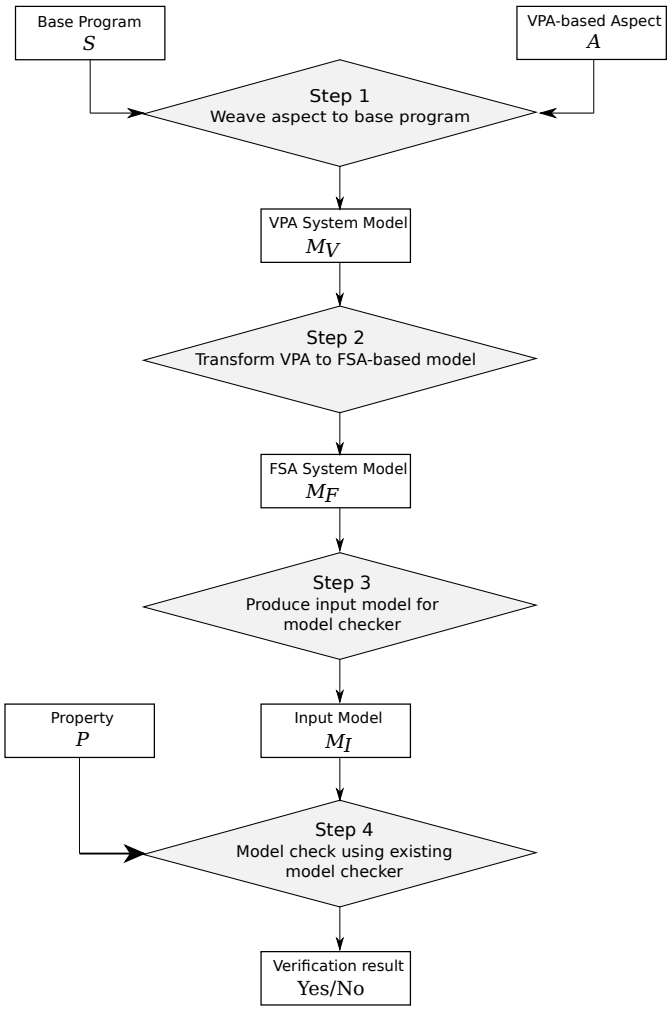


Figure 7.1: Model checking procedure on VPA-based AO programs

The procedure consists of four major steps:

1. **Weaving**: Weave aspect A into base program S to obtain VPA M_V. This weaving process is very similar to the one defined in Section 4.3.3 of Chapter 4. Basically, in this process, we construct the VPA M_S that represents the base program S and the VPA M_A that represents the pointcut of aspect A. Matching M_A against the execution of M_S allows us to determine applicable join points where aspect advice should be executed. We then add states and transitions representing the aspect advice to the VPA M_S at applicable join points to obtain the composed VPA model M_V. Figure 7.2 shows an example of VPAs representing a base program, an aspect and the composed system. In this example, aspect advice ad is supposed to be executed at the join point represented by the VPA return transition $\underline{m}$. Therefore, the composed model M_V includes additional state and transition representing this advice.

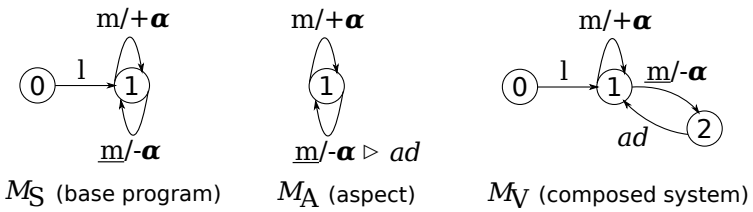


Figure 7.2: VPAs of a base program, an aspect and the composed system

This weaving works as we can statically build the woven program from the base program and a VPA-based aspect. The VPA-based aspect language currently does not allow dynamic conditions in pointcuts so we always know in advanced where the aspect will be applied.

2. **Abstraction**: Transform VPA M_V into an abstract model, denoted as M_F, that can serve as an input model for a typical model checker. This step is indeed an abstraction process because we have to approximate some features that supported by a VPA but not by a FSA, most importantly, the stack.

3. **Input model generation**: Produce an input model M_I for a specific model checker. Since we aim at a framework that is as independent from a real model checker as possible, the specification of the abstract model M_F is quite general, *i.e.,* not defined in the language of any specific model checker. Hence, after obtaining M_F we have to produce a description of the model in the language accepted by the model checker that we actually use. This step can be automatically performed by a converter.

4. **Check**: Verify whether the model M_I satisfies property P using the model checker. This step is automatically performed by the chosen model checker.

7.3.2 Abstracting VPAs into finite-state machines

We now present the details of the second step where we construct an abstract model from the original VPA-based system model in the following.

Every VPA features a stack whose content can be updated by transitions. Transitions of a VPA are classified into three different groups: local transitions that do not affect the stack, call transitions that push symbols onto the stack, and return transitions that pop symbols from the stack. An input model for a typical model checker, however, does not feature a stack. Therefore, we have to eliminate the stack of a VPA system model in order to transform it into a model that can be accepted by a model checker. The resulting model should approximate sufficiently precisely the relevant behaviors of the original model so that interesting verification results for the original VPA model can be obtained from the resulting model.

We basically have two options for handling the stack of a VPA. The first option is to approximate the depth of the stack to a given value so that we can use a limited number of finite-state transitions to model nested structures. The second option is to simulate the stack and associated constraints using other means such as variables and conditions defined for transitions. The goal of both options is to retain the semantics of the stack as much as possible. In the following we discuss the two approaches in more details.

7.3.2.1 Approximating the depth of the stack component of a VPA

The stack component of a VPA permits the modeling of nested structures of opening and closing events. Normally, there is no restriction on the depth of the stack component of a VPA and thus nested structures described by a VPA can include arbitrary number of sub-levels. Without a stack, a typical FSA cannot describe such kinds of nested structures. However, if we know in advance the maximum depth of a nested structure, it is possible to model the structure by a FSA. In principle, this FSA is constructed by composing a number of "sub-FSAs" that match the nested structure until the given maximum depth. In the following we introduce the general ideas of such abstraction approach where we approximate the depth of the stack component of a VPA to a given number so that we can use a FSA to model the system originally defined by that VPA.

Let us consider an example in order to demonstrate the above abstraction method. In this example, we wish to transform the VPA modeling the nested structure of the opening event m and the corresponding closing event $\underline{m}$. Figure 7.3 illustrates the original VPA M_V and three different FSAs obtained by approximating the maximum depth of M_V'stack. The stack symbol α is pushed into the stack when m is taken and popped off the stack when $\underline{m}$ is taken. State 0 of M_V is the final state (*i.e.,* accepting state). We then abstract the VPA M_V by approximating the depth of the stack of M_V to three values 1, 2, 3 and obtain three FSAs M_F^1, M_F^2, M_F^3 respectively. In other words, M_F^1 defines all the nested structures that can be defined by M_V when the maximum depth of the stack is 1 and states $\{0, 1\}$ are the accepting states. Similarly, we obtain M_F^2, M_F^3 (and others) when we increase the number to which we approximate the depth of stack of M_V.

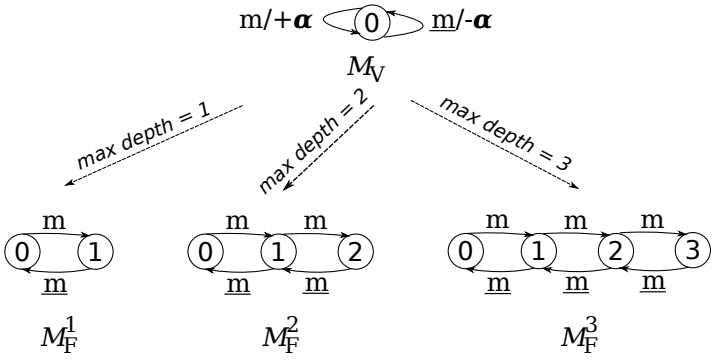


Figure 7.3: Approximating the depth of the stack of a VPA

This abstraction method is an under-approximation method. That means, the structures (or sequences of events) described by the obtained FSA are the subset of those described by the original VPA. Hence, if a model checker proves that the system described by the obtained FSA violates a property ϕ then we can conclude that the original VPA also violates ϕ. The precision of this abstraction method depends on the value of the depth of stack that we choose to serve as the maximum value. The higher the depth the more precise the abstraction.

7.3.2.2 Simulating the stack component of a VPA

In this approach, we perform a finer abstraction on a VPA model that yields a variant of a finite state automaton that carries additional information from the stack of the original VPA

model in addition to the standard state and transition sets. The resulting model is more expressive than a finite state automaton: we refer to this model as a counter-based transition system. Note that although counter-based transition systems are more expressive than FSA, they still allows model checking performed by model checkers that support them.

Definition 5 (Counter-based transition system)**.** *An intermediate counter-based transition system, which is a variant of the labeled transition system introduced by Milner [93], is a 5-tuple* $M = (Q, Q_{in}, \delta, \chi, Q_F)$*, where*

- Q *is a set of states*
- Q_{in} *is a set of initial states*
- Q_F *is a set of final states*
- χ *is a set of counters that are initiated by constant expressions*
- $\delta \subseteq (Q \times Q \times L \times A \times C)$ *is a transition relation where*
  - L *is a set of labels*
  - A *is a set of actions associated to a transition. Actions are defined by assignments that increase or decrease counters.*
  - C *is the set of conditions (or guards) associated to a transition. Conditions are defined by inequalities that express whether counters are greater or smaller than other counters or constant expressions.*

The principle behind the transformation of VPAs into these counter-based transition systems is to simulate the stack component and transitions of a VPA model by adding a set of counters, actions, and conditions associated to transitions of a finite state automaton. In the resulting counter-based transition model, counters are used to keep track of the number of call and corresponding return transitions. A transition can only occur if its condition (or guard) is satisfied at the current state. When a transition occurs, the action associated to that transition is also taken which will then modify its counter. Hence, this transformation actually produces a variant of FSA which is more expressive than a "pure" FSA. However, most of the currently available model checkers accept a similar model which is more expressive than a FSA (but not as expressive as a VPA since such model normally does not have a stack).

Definition 6 (Construction of counter-based transition system from a VPA)**.** *From a VPA* $V = (Q_V, Q_{in_V}, \Gamma_V, \delta_V, Q_{F_V})$*, we construct a counter-based transition system* $F = (Q_F, Q_{in_F}, \delta_F, \chi_F, Q_{F_F})$ *whose state and counter sets defined as follows:*

$$
\begin{aligned}
Q_F &= Q_V \\
Q_{in_F} &= Q_{in_V} \\
Q_{F_F} &= Q_{F_V} \\
\chi_F &= \Gamma_V
\end{aligned}
$$

The transition set δ_V is a one-to-one mapping on the transition set δ_F while the stack of the VPA is simulated by the use of counters and conditions in F. These counters and transition conditions ensure that the number of return transitions never exceeds the number of corresponding call transitions. The transition set δ_F is created from the transition set δ_V as follows:

- For each push transition $ct = (q, a, q', \gamma)$ of VPA model V, we create a transition $at = (q, q', a, \gamma' = \gamma + 1)$.
- For each pop transition $rt = (q, a, \gamma, q')$ of VPA model V, we create a transition $at = (q, q', a, \gamma' = \gamma - 1, \gamma > 0)$.
- For each local transition $lt = (q, a, q')$ of VPA model V, we create a transition $at = (q, q', a)$.

The above approach for stack simulation is, however, only applicable to a limited class of VPA models. When there is one pair of call/return (or open/close) events in the VPA model, the simulation reflects the stack and constraints precisely, *i.e.,* no information is lost through abstraction. When there are more than one pair of call/return events, the above simulation does not always reflect the stack and constraints correctly because the use of counters for different pairs of events does not record explicitly the stack content.

Let us consider a small example that illustrates the limitation of stack simulation. Figure 7.4 demonstrates an example where the resulting counter-based transition model contains scenarios which do not exist in the original VPA model. The original VPA model describes a system consisting of nested calls and returns of m and n. The pair of events $m, \underline{m}$ push and pop α while $n, \underline{n}$ push and pop β in and out of the stack. At state 2, we expect that a sequence of returns $\underline{n}$ will occur before a sequence of $\underline{m}$ since β symbol(s) would be on the top of the stack when the system first evolves to this state. However, the resulting counter-based transition model does not reflect this property. It implies that both $\underline{m}$ and $\underline{n}$ can occur at state 2 without any constraint on the order of occurrence.

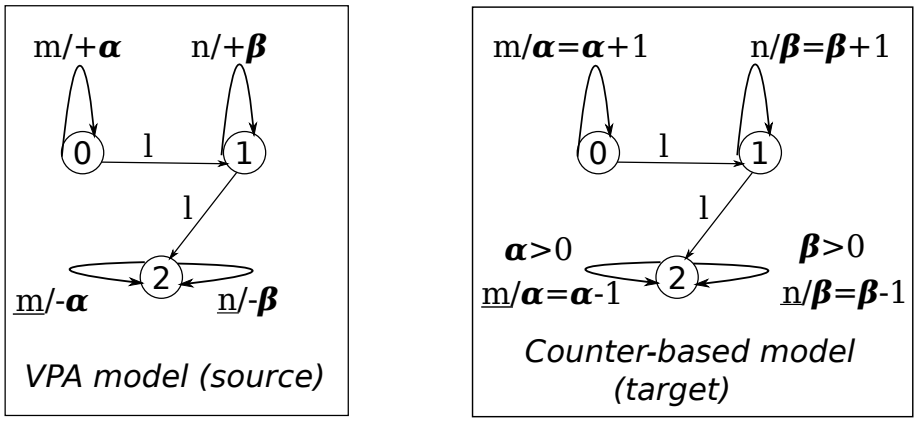


Figure 7.4: Stack simulation with counters and conditions

Hence, the use of variables to simulate the stack content is only correct when the transitions in the model push or pop using the same stack symbol. Although solutions to remedy the above problem exist for some situations (for instance, we can add condition $\beta == 0$ to

the $\underline{m}$ transition at state 2), we try to avoid creating such models (if possible) by doing stack simulation on counterpart models where only one stack symbol is used.

Similarly to the first abstraction approach, creating an abstract model by stack simulation is also an under-approximation of the original model because the value of the variable representing a stack always has an upper bound (which may, however, be quite large).

As we have just shown above, both approaches to transform a VPA model to a counter-based transition model involve some approximation of the original model. As a consequence, verification results achieved using model checking techniques over the FSA model may not always hold for all original VPA models. If the model checker proves that the abstract model satisfies the property, we can only conclude that the original model satisfies the property on the condition that the depth of the stack is within a given range. In many cases, this conclusions is sufficient. On the other hand, if the model checker proves that the abstract model violates the property, we can use the counter-examples provided by the model checker as a clue to investigate the violation and improve the original model.

7.4 Comparison of existing model checkers

In order to realize the framework above we have investigated several model checkers. Table 7.1 presents a list of a few well-known model checkers, their features that are particularly important to our study, and references to articles about them.

Table 7.1: Some model checkers and corresponding properties

Model checker	System model	Support for counters	Property	References
SPIN	Promela (SPIN's language)	Yes	LTL	[65, 64]
NuSMV	Finite state machine	Yes	CTL,LTL	[38, 37]
UPPAAL	Real-time automata	Yes	CTL	[78, 26]
LTSA	Finite state machine	No	CTL	[87]

Three properties of model checkers we are particularly interested in are the following:

- The model of the input system
- What types of properties are supported
- Efficiency and usability of the model checkers

Input system model. From our point of view, the type of input model that is supported by a model checker is the most important criterion that determines our choice of model checker. We are especially interested in model checkers that support the use of counters, guards and assignments associated to transitions as these characteristics facilitate the transformation from the abstract counter-based transition model to the input model for the model checker. All model checkers in Table 7.1 use input models that are given as textual descriptions of state machines in their tool-specific formats. SPIN, NuSMV and UPPAAL permit the declaration of variables, guards and assignments. The state machines used by these model checkers are relatively close to the abstract counter-based system model that is generated as part of our approximation process.

Property specification. Generally, most of the model checkers support properties specified in some variants of LTL or CTL logics. SPIN and LTSA accepts LTL properties. NuSMV provides the most flexibility by accepting both LTL and CTL properties. UPPAAL accepts only CTL properties. Although the type of logic in which the property is specified is not the most important criterion in our context, we prefer model checkers that support CTL properties as we might want to verify properties that involve different execution paths.

Efficiency and usability. Since the abstraction from VPA-based properties to regular systems generates large finite-state machines (that are of a very specific form), the efficiency of model checkers is an important criterion for the feasibility of our approach. However, while the first two features, system and property specification of model checkers, can be evaluated simply, it is more difficult to have a comparative view on the scalability properties of model checkers since verification tools are typically designed and optimized for different specific domains. Usability is another less important criterion that we have taken into account. We are basically interested in how easy it is to create system models and whether a model checker provides simulation tools. We have found that there are sufficient means to guide the modeling with SPIN, NuSMV and UPPAAL while there is almost no documentation for LTSA available. Both SPIN and UPPAAL provide GUI-based editors that provide intuitive support for the creation of system models. These two model checkers also provide simulation tools that facilitate modeling and testing processes.

7.4.1 UPPAAL for model checking of VPAs

We have selected the model checker provided by the UPPAAL tool [78] as the model checker to conduct the experiments. The UPPAAL model checker basically supports the verification of finite state automata (more precisely, a kind of timed finite state automata).

In principle, all the aforementioned model checkers are capable of verifying some variants of finite state automata. Our choice of UPPAAL to conduct our validation is justified by two main reasons. First, UPPAAL allows the definition of variables and constraints, both of which can be associated to transitions in a system model. This ability allows us to build the model using variables and constraints to simulate the stack component of an original VPA model. Second, UPPAAL provides the convenience of an integrated tool environment where we can construct the model and run simulations interactively. This feature is important in practice because it enables an iterative development process for the definition of models that are correct and amenable to verification, a process that can be highly time and resource consuming depending how the model is constructed. There have been times when we had to change the way we model the system in order to avoid deadlock situations due to incorrect modeling or to put a lower upperbound (to the default upperbound) to a variable in order for the verification process to terminate.

7.5 Model-checking VPA-based P2P systems

In this section, we present two validation scenarios to demonstrate how model checking is performed using the VPA approximation method presented in the previous section. We use the P2P-based grid system introduced in Chapter 5 as the base application for our validation scenarios.

We have developped two validation scenarios to which we have applied UPPAAL in order to verify the collaboration regarding task submission and distribution among different parties in the P2P-based grid system, *i.e.,* task submitters, managers and workers. When the protocols that govern the communication among parties contain sequences of several synchronized events, it can be difficult to manually evaluate the correctness of the collaboration especially when there are more than two parties involved. Therefore, the model checker can assure us of the smooth cooperation or detect possible conflicts among parties in the system. In the first scenario, we evaluate the collaboration between a task submitter and a manager in which the task submitter sends tasks to the manager and then gets back results from the manager. In the second scenario, we extend the first evaluation by considering the third party, the worker, in the collaboration. In this case, the manager communicates with the worker regarding task distribution and result collection.

7.5.1 Validation scenario 1: collaboration between two parties

In this example, we consider the cooperation between a task submitter and a manager according to the communication protocols illustrated in Figure 7.5. The protocols we consider here are slightly different from the ones presented in Chapter 5. We have removed internal events that do not participate directly in the communication as they do not have any effect in the correctness of the collaboration.

The protocols describe the transitions in a communication session as follows. From the initial states, both parties enter the communication by the *start* events. Next, the submitter can send tasks (described by the *sendTask* event) to the manager while the manager is in the position to accept the tasks from the submitter. After receiving all the tasks and having the tasks getting done, the manager sends back a message that contains the results of the tasks (described by the *sendResult* event). There is a constraint imposed on the *sendTask* and *sendResult* events: the number of *sendResult* events cannot exceed the number of *sendTask* events. Therefore, we model *sendTask* events as *call* transitions and *sendResult* events as corresponding *return* transitions. After the submitter receives the results, the submitter and the manager can return to their initial states which are also their final states. The composition of models of the submitter and the manager is considered the base program S, cf. the framework definition illustrated in Figure 7.1.

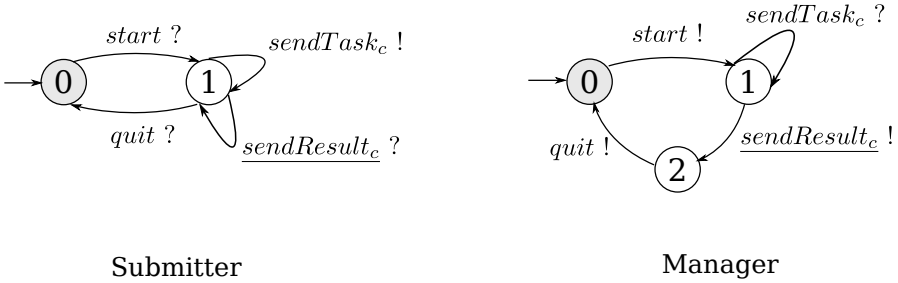


Figure 7.5: Protocols of the submitter and the manager

Now let us assume that the submitter expects to receive a separate result message for each of the task it has sent to the manager. In the original protocol of the manager, the manager only sends results back to the submitter through only one *sendResult* message event. Hence, we have to modify the protocol of the manager so that it can send several messages to return

results to the submitter. Note that any modifications to the protocol of the manager has to meet the constraint which requires the number of *sendResult* events not to exceed the number of *sendTask* events. This modification can be done using a VPA-based aspect with a *closeOpenCalls* advice defined as follows:

$$A = \mu a.start\ ;\ \mu b.sendTask_c\ ;\ b \\ \square\ \underline{sendResult_c} \triangleright closeOpenCall_{\underline{sendResult_c}}\ ;\ quit\ ;\ a$$

The above aspect A will add a number of *sendResult* events to the original protocol of the manager so that the manager sends an equal number of *sendResult* message back to the submitter and thus satisfies the expectation of the submitter on the protocol level. Given the base program S presented above and the above aspect A, we perform Step 1 of the model checking procedure shown in Figure 7.1 and weave the aspect A with the base program S and obtain the VPA-based system model M_V illustrated in Figure 7.6.

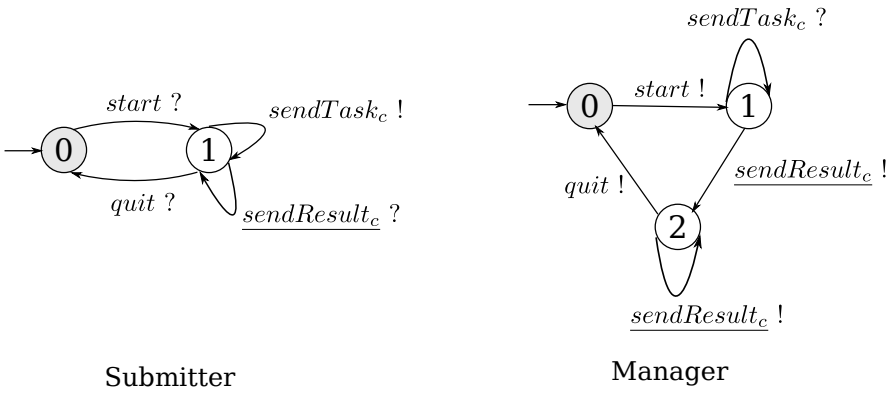


Figure 7.6: Models of submitter and manager after modified by aspect A

In Figure 7.6, the protocol of the submitter does not change while the protocol of the manager is modified so that there is an extra transition at state 2. This additional transition represents *sendResult* events generated by aspect A. After obtaining the VPA-based system model M_V, we proceed to take Step 2 and Step 3 of the chart shown in Figure 7.1 to transform this model into an input model for model checker UPPAAL. Since we only plan to use the model checker UPPAAL to do the verification, we skip Step 2 (transforming the VPA-based model to a generic FSA model) and build the input model for UPPAAL directly from the VPA-based system model (Step 3).

As discussed in the previous section, we can use one of the two abstraction methods to construct the input model for the model checker. We choose to apply the second abstraction method (*i.e.,* simulating the stack component) to the VPA-based system model because UPPAAL supports the use of variables and constraints for (fixed-numbered repetitions of) transitions.

Figure 7.7 and Figure 7.8 show the protocols the submitter and the manager that have been adapted to be input models for model checker UPPAAL respectively.

In the model of the submitter shown in Figure 7.7, we use the variable *tc* to simulate the stack manipulated by the *sendTask* and *sendResult* transitions. When the transition representing the *sendTask* event takes place, we increase the *tc* variable by one. When the

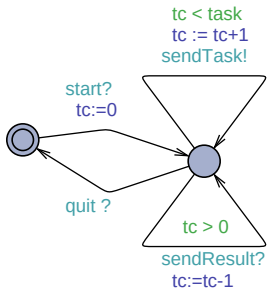


Figure 7.7: Protocol the submitter modeled in tool UPPAAL

transition representing the *sendResult* event takes place, we decrease the *tc* variable by one. We use the constant *task* (which can be set as a parameter for an instance of the submitter) to mark the upper bound value of the *tc* variable. In fact, this constant defines the maximum depth of the stack. We define the constraint $tc < task$ for the transition representing *sendTask* so that no transition can be taken if the number of *sendTask* events reach the maximum allowed. Similarly, we use constraint $tc > 0$ on the transition representing *sendResult* so that the number of *sendResult* never exceeds the number of *sendTask* which is the constraint implied by the stack in the original VPA-based model.

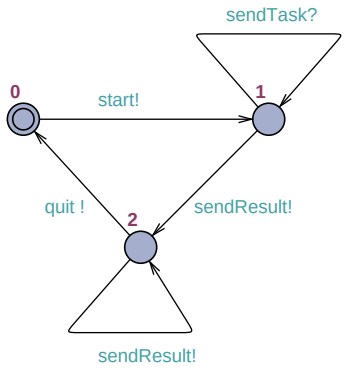


Figure 7.8: Protocol of the manager modeled in tool UPPAAL

In the model of the manager shown in Figure 7.8, we do not have to use other variables and constraints to simulate the stack because we already do that for the submitter model and that communications between processes in UPPAAL are synchronized. If the submitter is not in the state where it can receive *sendResult* messages, then the transitions representing *sendResult* events in the manager model cannot take place.

We then apply step 4 in the model checking procedure and use UPPAAL to verify the input model. We would like to know whether the new system after using aspect A to modify the manager protocol can run smoothly without any possible deadlock. This property is encoded in the UPPAAL property specification language as $A[\ ]$ *not deadlock*, *i.e.,* there is no path in the system where deadlock can occur. We have successfully used UPPAAL to verify the above model against this property. The model checker could confirm that the property was satisfied by the input model in just a few seconds.

7.5.2 Validation scenario 2: collaboration among three parties

In this example, we consider a more complex setup of the communication in the peer-to-peer grid computing system. Figure 7.9 shows the protocols of three parties in the system: the submitter, the worker, and the manager. While the protocol of the task submitter does not change in comparison with the protocol shown in example 1, the protocol of the manager has changed and there is an additional protocol that represents the participation of the worker in the communication.

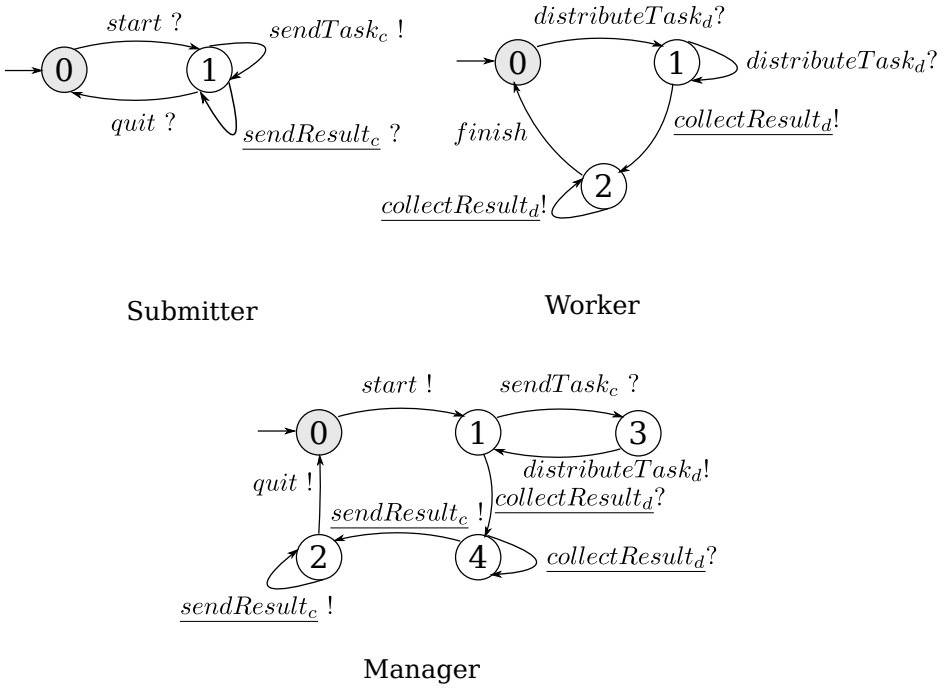


Figure 7.9: Protocols of the task submitter, the worker, and the manager

The protocol of the worker shown in Figure 7.9 expresses the transition between the states of the worker as follows. From the initial state, the worker waits for task distribution (represented by *distributeTask* events) from the manager. The worker can receive several tasks. The worker then starts to process the task, *i.e.,* performing the utility computation. Task processing is considered as being a local event of the worker, so we omit it from the protocol for simplicity. The worker then returns the results of the tasks to the manager through *collectResult* events. The number of the *collectResult* events should not exceed the number of *distributeTask* events so we model *distributeTask* as *call* transitions and *collectResult* as *return* transitions. After returning all the results to the manager, the worker finishes its session by turning back to its initial state.

The protocol of the manager shown in Figure 7.9 expresses the transition between the states of the manager as follows. From the initial state (state 0), the manager starts its session and enters state 1. At this state, the manager can start accepting requests to process tasks from the task submitter (represented by *sendTask* events). After receiving a request, the manager distributes the task to its workers (represented by *distributeTask* events). It can continue to wait for other requests and to distribute them. The manager then begins to collect results from the worker (by entering state 4). After collecting all the results, it sends

the results to the task submitter and finally quits the session.

Hence, in this example, the base program S is the combination of the three transition systems shown in Figure 7.9. Now let us assume that we would like to add the possibility for the worker to quit the computing session in the middle of the session, *i.e.,* before some of the results have been collected. We would implement this modification by an aspect defined as follows:

$$\begin{aligned} A = & \mu a.distributeTask \ ; \ a \\ & \square \ quit \triangleright finish \ ; \ a \end{aligned}$$

The application of the above aspect A to the protocol of the worker changes the original protocol to the one illustrated in Figure 7.10. The new transition from state 1 to state 0 represents the application of the aspect.

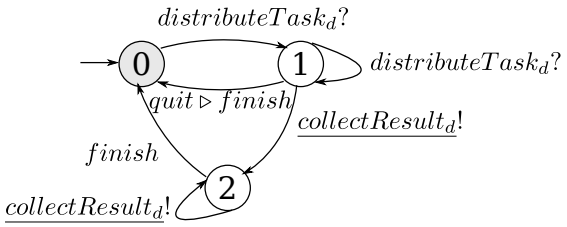


Figure 7.10: Protocol of the worker after the application of aspect A

Hence, the system model M_V obtained by aspect weaving according to the chart in Figure 7.1 is the combination of three protocols: the protocols of the task submitter and the manager in Figure 7.9, and the new protocol of the worker in Figure 7.10. Similar to example 1, here we skip step 2 of the model checking procedure and build the input model for UPPAAL directly from M_V.

Figure 7.11 and 7.12 show the UPPAAL model of the manager and the model of the worker respectively (the UPPAAL model of the submitter is the same as the one shown in Figure 7.7). The input model M_I is the combination of these three models.

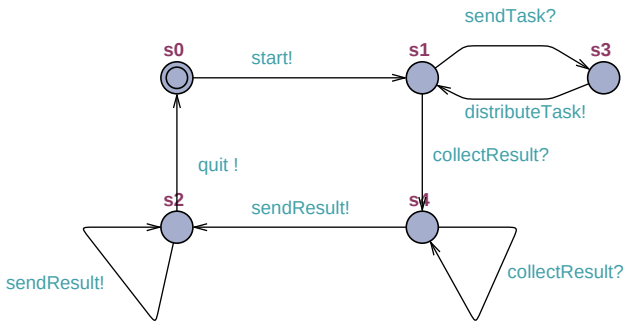


Figure 7.11: Protocol of the manager modeled in tool UPPAAL

In the model of the worker presented in Figure 7.12, we use variable at to simulate the stack that is manipulated by *distributeTask* and *collectResult* transitions. We put constraint $at > 0$ on the *collectResult* transitions to ensure that the number of results returned does not exceed

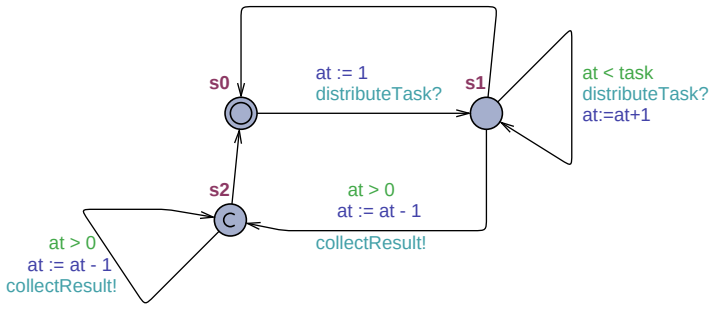


Figure 7.12: Protocol of the worker modeled in tool UPPAAL

the number of tasks distributed. The transitions representing $finish$ events are modeled as ϵ-transitions in UPPAAL since it is not possible to have not synchronized transitions in the UPPAAL models.

Now we again verify the deadlock property on the input model M_I. The UPPAAL model checker proves that the deadlock property is not satisfied by the input model M_I that comprises three protocols of the submitter, manager, and worker. Hence, the modification of the system by the aspect breaks the deadlock property of the system. Note that UPPAAL enables the deadlock satisfaction proof on the original base program, *i.e.,* without the aspect modification satisfies the deadlock property.

The problem introduced by the aspect happens when the following conditions are met:

- The submitter is in state s_1 where it cannot send more tasks to the manager because it reaches the maximum number of tasks it can send.
- The manager is in state s_1 where it waits to collect results from the worker.
- The worker is in state s_0 where it waits to get tasks distributed by the manager.

When the submitter is in state s_1 and it cannot send more tasks, it can only take the *sendResult* transition to proceed. However, it has to wait for the manager to take the corresponding transition. When the manager is in state s_1, it can take the *sendTask* transition or the *collectResult* transition to proceed. Since, the submitter cannot send more tasks, the only way to the manager to get out of state s_1 is to collect result from the worker. However, at the moment, the worker has already quit the previous session, currently is in its initial state s_0 and waits for new tasks which cannot be provided by the manager. The whole system blocks at this point.

7.6 Conclusions

In this chapter, we have discussed potential model checking approaches for verifying VPA-based models. Since there exists no model checker that explicitly supports VPA input models, we have chosen to use existing model checkers to verify finite-state based abstractions of VPA-based models. We have introduced a model checking procedure for the verification of VPA-based aspect systems. This procedure includes a weaving step where aspects are weaved into the base program, the abstraction steps where VPA models are transformed

into more abstract models (basically variants of FSAs) that are supported by existing model checkers, and the verification step where the interested property is verified by the chosen model checker. However, this model checking procedure has two limitations. First, we still have to run verifications on the woven system, which means state space explosion is possible if the system model is large. Second, we verify the abstract system rather than the actual system so the verification result is only reliable provided that certain conditions (depending on the abstraction method employed) are met.

We have introduced two abstraction methods that can be employed to create a finite-state based input model from a VPA-based one. In the first method, we basically fix the depth of the nested structure then unfold the repetitions into sequences of events. In the second method, we introduce the concept of counter-based transition system where we use variables and constraints (or also called guards) that are supported by the model checker to simulate the stack of the VPA. These two abstraction methods are subject to the same limitations. First, they require the restriction of the depth of stack-based operations. Second, they do not apply well for protocols that involve two or more pairs of corresponding events nesting within the others.

Finally, we have showed how model checking is done using our approach through the verification of two validation scenarios with model checker UPPAAL.

The limitations of our current model checking approach suggest a few directions for future work. First, the modular model checking technique [59] could be attempted for VPA-based aspects. Second, a tool that systematically transforms a VPA model to a counter-based transition model can be useful as the transformation process is very tedious and error-prone if the original system is not trivial. Such a tool may enable a semi-automatic abstraction process and improvement steps suppressing obvious non-existing paths in the original model are performed manually.

Chapter 8

Conclusion

The main goal of this thesis consists in the study and development of an expressive aspect-oriented language that is amenable to property analysis and verification. We have proposed using VPAs as a foundation for the definition of the VPA-based aspect language that supports expressiveness and property verification. In addition, we have studied its application and analysis support in the context of component composition. Finally, we have considered the use of model checking technique for the verification of VPA-based aspect systems.

The remainder of this chapter is organized as follows. Section 8.1 recapitulates the major contributions of this thesis. Section 8.2 presents directions for future work.

8.1 Contributions

We have presented the following contributions in this thesis:

VPA-based aspect language and VPAlib. We have defined the VPA-based Aspect Language, a new history-based aspect language defined upon the class of visibly pushdown automata. Our aspect language features a VPA-based pointcut language that is capable of properly expressing non-regular protocols, including well-balanced nested pairs of events or recursive function calls. In contrast to aspect languages that feature context-free or Turing-complete pointcuts, our VPA-based aspect language allows more possibilities for analyzing the properties of the aspects and/or the program modified by the aspects.

We have provided, in particular, support for for non-regular nesting structures, *depth* constructors, *permutation* operators, and regular expressions. These special constructors help facilitate the declarative definition of VPA-based pointcuts. We have also introduced the *closeOpenCall* operator as a special advice operator that can be used to close an open calling context. We have defined a formal framework that allows to describe the semantics of our aspect language precisely, *e.g.,* in order to provide a guidance on the implementation of VPA-based aspects. More concretely, this semantic framework defines how we can construct a VPA from a VPA-based aspect , match the VPA against a base program execution, and weave the corresponding advice into the base program.

We have implemented *VPAlib*, a library that provides the implementation of essential data structures and operations for the VPA. The VPAlib library has been implemented in Java SE 6 and released under LGPL license. We have provided a set of closure operations, the

determinization operation and inclusion check for VPAs. These operations are critical for the construction of VPAs as well as the verification of VPA-based protocols and aspects.

Analysis of VPA-based aspects and application to the evolution of component-based systems. We have introduced an analysis technique for the detection of potential interactions among VPA-based aspects that could may occur due to nondeterministic weaving. We calculate the product automaton of two VPAs and pickup simultaneous occurences of the same transitions in both VPAs as potential interactions. Our analysis technique is practically supported by the interaction operation implemented in our VPAlib library.

Furthermore, based on the formal properties of VPA, we have developed constructive means for the construction and evolution of correct component-based systems. We have introduced an approach to prove the preservation of properties of systems modified by VPA-based aspects. Our approach exploits the characteristics of our aspect language in order to analyze the properties of a system. Concretely, we have studied the characteristics of three aspect advice operators: (i) operators to close open call events, (ii) operator to cut the depth of sequences of events, and (iii) operator to insert pairs of events. We have formally proved that substitutability and compatibility properties are preserved for certain classes of component interaction protocols and aspects that employ one of these three advice operators. Our approach enables the preservation of correctness properties to be proved for classes of aspects and protocols rather than individual ones

Harnessing model checking tools for VPA-based aspects. In order to harness the efficiency of modern model checking tools, we have designed a framework for the use of model checking techniques for the verification of VPA-based systems. VPA-based system models are first abstracted into simpler input models that are practically supported by an existing model checker. We then run the model checker tool to simulate and verify the property represented by the input model. We have demonstrated our model checking framework through two examples in the context of the peer-to-peer grid computing system. Model checking and simulating using the checker tool has allowed us to discover errors in the system models that can lead the systems into deadlock situations.

Applications. We have also shown how VPA-based aspects can be applied in two application systems: (i) a typical setup of some remote access systems, and (ii) a grid computing system over peer-to-peer network. In the first system, we use VPA-based aspects to supervise access in nested login sessions. In the second system, we use VPA-based aspects to implement modules handling task monitoring, canceling and distributing in a typical grid computing system.

8.2 Perspectives

This thesis work paves the way for a number of directions for future work:

VPAL. The aspect language should be made more expressive. This includes a larger set of pointcut constructors, VPA-based mechanisms as well as other mechanisms proposed for regular and non-regular protocols. Similarly, it is useful to have a more powerful advice language with more specific operators. Second, the VPA-based aspect language should be

implemented for usage as an full-fledged development means. A long-term goal of future work consists in the integration of VPA-based aspects with other types of protocols.

As to the analysis and verification of VPA-based aspects, operations such as the determinization and inclusion check are critical. However, our current implementation provides only limited optimizations and is subject to some severe limitations regarding the size of models that can be reasonably handled. Hence, better optimization of implementations for non-regular protocols should be investigated.

VPA-based aspects for components. Apart from compatibility and substitutability properties, we could aim at proving the preservation of other properties using the approach proposed in chapter 6. Besides, more classes of aspects and protocols should be studied so that more classes that help establish a larger set of preservation properties can be defined.

Model checking. The performance and efficiency of model checking using different model checkers might be quite different. Hence, one should evaluate usage of other model checkers than UPPAAL, such as NuSMV or Spin, with the model checking framework that we have introduced in chapter 7. Other approaches to use model checking techniques for VPA-based systems could also be considered. For instance, we can use the operations provided by the VPAlib library to perform analyses directly on VPA models without having to compromise the precision of the verification results.

Applications. There is a large wealth of application domains that may benefit from more expressive interaction protocols and provably correct evaluation methods based on protocols. Service choreographies constitute but one important example. The development of aspect languages that allow to express and manipulate software systems in terms of their interaction protocols are of major futur interest and importance.

Bibliography

[1] *abc*. The AspectBench Compiler. `http://aspectbench.org`.

[2] *Beingrid Project Home Page*. `http://www.beingrid.eu/`.

[3] *CAPE Home Page*. `http://www.cs.technion.ac.il/ ssdl/research/cape/`.

[4] FIPA interaction protocol specifications.

[5] *General formulation of the Law of Demeter*. `http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html`.

[6] *Gnutella Message Board*. `http://www.gnutellaforums.com/`.

[7] *JBoss.org*. `http://www.jboss.org/`.

[8] *JXTA technology*. `https://jxta.dev.java.net/`.

[9] *Kazaa Home Page*. `http://www.kazaa.com/`.

[10] *Napster Home Page*. `http://www.napster.com`.

[11] *Remote Desktop Protocol*. `http://www.rdesktop.org/`.

[12] *Spring Framework*. `http://www.springframework.org/`.

[13] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In *ECOOP*, pages 372–395, 1992.

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

[15] Mehmet Aksit and Anand Tripathi. Data abstraction mechanisms in sina/st. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 267–275, New York, NY, USA, 1988. ACM.

[16] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *ICSE*, pages 187–197. ACM, 2002.

[17] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, et al. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2005.

[18] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.

[19] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications*. Springer-Verlag, Berlin, 2004.

[20] Rajeev Alur. Model checking: From tools to theory. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 89–106. Springer, 2008.

[21] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 202–211, New York, June 13–15 2004. ACM Press.

[22] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETIhome: An experiment in public-resource computing. *Comm. ACM*, 45(11):56–61, November 2002.

[23] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[24] Paul Attie, David H. Lorenz, Aleksandra Portnova, and Hana Chockler. Behavioral compatibility without state explosion: Design and verification of a component-based elevator control system. In I. Gorton et al., editor, *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, number 4063, pages 33–46, 2006.

[25] Steffen Becker, Sven Overhage, and Ralf Reussner. Classifying software component interoperability errors to support component adaption. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings*, pages 68–83. Springer, 2004.

[26] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[27] Robert Pawel Bialek. The architecture of a dynamically updatable, component-based system. In *COMPSAC*, pages 1012–1016. IEEE Computer Society, 2002.

[28] Francisco Vilar Brasileiro, Eliane Araújo, William Voorsluys, Milena Oliveira, and Flavio de Figueiredo. Bridging the high performance computing gap: the ourgrid experience. In *CCGRID*, pages 817–822. IEEE Computer Society, 2007.

[29] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. pages 349–360, 1987.

[30] Alan W. Brown and Kurt C. Wallnau. Enginnering of component-based systems. In Alan W. Brown, editor, *Component-Based Software Engineering*, pages 7–15. IEEE Press, 1997.

[31] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in java. *Software, Practice Experience*, 36(11-12):1257–1284, 2006.

[32] C. A. R. Hoar. *Communicating Sequential Processes*. Prentice Hall, 1985.

[33] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *ICSE*, pages 285–294. IEEE Computer Society, 2001.

[34] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with run-time types for the java programming language. *SIGPLAN Not.*, 33(10):201–215, 1998.

[35] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis for interrupt-driven programs. *Inf. Comput.*, 194(2):144–174, 2004.

[36] C. C. Chiang. The use of adapters to support interoperability of components for reusability. *Information & Software Technology*, 45(3):149–156, 2003.

[37] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[38] Alessandro Cimatti et al. NuSMV2: an opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer-Verlag, July 27–31 2002.

[39] E. M. Clarke, E. Allen Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[40] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[41] ObjectWeb Consortium. *Fractal BPC*. `http://fractal.objectweb.org/fractalbpc/`.

[42] ObjectWeb Consortium. *Julia - the reference implementation platform of Fractal component model*. `http://fractal.objectweb.org/julia/`.

[43] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In Marcello Bonsangue and Einar Broch Johnsen, editors, *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31. Springer, 2007.

[44] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[45] Giovanni Denaro and Mattia Monga. An experience on verification of aspect properties. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 186–189, New York, NY, USA, 2001. ACM.

[46] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In Dave Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.

[47] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of GPCE'02*, LNCS 2487, pages 173–188. Springer Verlag, October 2002.

[48] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, March 2004.

[49] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley Professional, September 2004.

[50] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In Kris Gybels, Maja D'Hondt, Istvan Nagy, and Remi Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, September 2005.

[51] Erik Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.

[52] Javier Esparza, Antonín Kucera, and Stefan Schwoon. Model-checking ltl with regular valuations for pushdown systems. In *TACS '01: Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software*, pages 316–339, London, UK, 2001. Springer-Verlag.

[53] Rod Johnson et al. *The Spring Framework - Reference Documentation*.

[54] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. *SIGPLAN Not.*, 37(5):13–24, 2002.

[55] Andrés Farías and Mario Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 2519 of *LNCS*, pages 995–1006, 2002.

[56] Andrés Farías and Mario Südholt. Integrating protocol aspects with software components to address dependability concerns. Technical Report 04/6/INFO, École des Mines de Nantes, November 2004.

[57] Fabrizio Gagliardi, Bob Jones, Mario Reale, and Stephen Burke. European DataGrid project: Experiences of deploying a large scale testbed for E-science applications. *Lecture Notes in Computer Science*, 2459:480–??, 2002.

[58] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[59] Max Goldman and Shmuel Katz. Maven: Modular aspect verification. In *TACAS*, pages 308–322, 2007.

[60] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 85–95, New York, NY, USA, 2007. ACM.

[61] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

[62] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD 04*, pages 26–35, 2004.

[63] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[64] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[65] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2003.

[66] Kohei Honda. Types for dyadic interaction. In *Proc. CONCUR '93*, number 715 in LNCS, pages 509–523. Springer, 1993.

[67] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, November 2000.

[68] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[69] *JBoss AOP*. `http://labs.jboss.com/jbossaop/`.

[70] T. Jensen, D. Le Mtayer, and T. Thorn. Verification of control flow based security properties. *Security and Privacy, IEEE Symposium on*, 0:0089, 1999.

[71] Yan Jin and Jun Han. Specifying interaction constraints of software components for better understandability and interoperability. In *Proceedings of ICCBSS 2005*, volume 3412 of *Lecture Notes in Computer Science*, pages 54–64. Springer, 2005.

[72] Shmuel Katz. Aspect categories and classes of temporal properties. *T. Aspect-Oriented Software Development I*, pages 106–134, 2006.

[73] Shmuel Katz and Marcelo Sihman. Aspect validation using model checking. In *Verification: Theory and Practice*, pages 373–394, 2003.

[74] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

[75] J. Kofron, J. Adamek, T. Bures, P. Jezek, V. Mencl, P. Parizek, and F. Plasil. Checking fractal component behavior using behavior protocols. presented at the 5th fractal workshop (part of ecoop06, 2006.

[76] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.

[77] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM.

[78] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[79] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[80] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Softw.*, 6(5):38–48, 1989.

[81] Karl J. Lieberherr and David Lorenz. Coupling aspect-oriented and adaptive programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented SoftwareLlm99 Development*. Addison-Wesley, 2004.

[82] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in demeter/java. In *ICSE*, pages 604–605, 1997.

[83] Karl J. Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[84] Juval Löwy. *COM and .NET component services*. Oreilly, sep 2001.

[85] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[86] N.A Lynch. *Distributed Algorithms*. 2006.

[87] Jeff Magee. Behavioral analysis of software architectures using LTSA. In *International Conference on Software Engineering*, pages 634–637. ACM, 1999.

[88] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, volume 989, pages 137–53, Sitges, Spain, 25–28 September 1995.

[89] Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, and Beth Stearns. *Applying Enterprise JavaBeans 2.1: Component-Based Development for the J2EE Platform*. Addison-Wesley Professional, may 2003.

[90] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[91] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD 03*, pages 90–99.

[92] Sun Microsystems. *JavaBeans Specification*, 1997. `http://java.sun.com/products/javabeans/docs/spec.html`.

[93] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[94] Robin Milner. A calculus on communicating systems. 92, 1980.

[95] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. 100(1):1–40, September 1992.

[96] Richard Monson-Haefel and Bill Burke. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., pub-ORA:adr, fifth edition, 2006.

[97] Ha Nguyen. *VPA library*. `http://www.emn.fr/x-info/hnguyen/vpa`.

[98] Oscar Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, 1993.

[99] Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.

[100] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Object-Oriented Series. Prentice-Hall, dec 1995.

[101] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192*, pages 73–80. Springer-Verlag, September 2001.

[102] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer Verlag, April 2005.

[103] Frantisek Plasil, Dusan Balek, and Radovan Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS'98)*, 1998.

[104] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *Transactions on Software Engineering*, 28(9), January 2002.

[105] K. V. S. Prasad. A calculus of broadcasting systems. In *Science of Computer Programming*, pages 338–358. Springer Verlag LNCS, 1991.

[106] Ralf Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *Proceedings of HICSS-34*. IEEE, 2001.

[107] Ralf H. Reussner. Counter-constraint finite state machines: Modelling component protocols with resource-dependencies. Technical report, School for Computer Science and Software Engineering, Monash University, VIC 3145 Australia, 2002.

[108] Joan Esteve Riasol and Fatos Xhafa. Juxta-cat: a JXTA-based platform for distributed computing. In Ralf Gitzel, Markus Aleksy, and Martin Schader, editors, *PPPJ*, volume 178 of *ACM International Conference Proceeding Series*, pages 72–81. ACM, 2006.

[109] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. *SIGSOFT Softw. Eng. Notes*, 29(6):147–158, 2004.

[110] David Rine, Nader Nada, and Khaled Jaber. Using adapters to reduce interaction complexity in reusable component-based software development. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 37–43, New York, NY, USA, 1999. ACM.

[111] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, New York, NY, 1997.

[112] James Sasitorn and Robert Cartwright. Component nextgen: a sound and expressive component framework for java. *SIGPLAN Not.*, 42(10):153–170, 2007.

[113] Heinz W. Schmidt and Ralf H. Reussner. Generating adapters for concurrent component protocol synchronisation. In *Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, 3 2002.

[114] Bran Selic. Protocols and ports: Reusable inter-object behavior patterns. In *ISORC*, pages 332–339. IEEE Computer Society, 1999.

[115] M. Sihman and S. Katz. Superimpositions and aspect-oriented programming. 2003.

[116] Marcelo Sihman and Shmuel Katz. Model checking applications of aspects and superimpositions. In Gary T. Leavens and Curtis Clifton, editors, *FOAL: Foundations of Aspect-Oriented Languages*, mar 2003.

[117] Therapon Skotiniotis, Jeffrey Palm, and Karl J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *ECOOP*, pages 477–500, 2006.

[118] Mario Südholt. A model of components with non-regular protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer Verlag, April 2005.

[119] John Sung and Karl Lieberherr. Daj: A case study of extending aspectj. Technical Report NU-CCS-02-16, Northeastern University, November 2002.

[120] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JasCo; an aspect-oriented approach tailored for component-based software development. In ACM Press, editor, *Proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29, March 2003.

[121] Clemens Szyperski, Domiinik Gruntz, and Murer Murer. *Component Software - Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2nd edition, 2002.

[122] Niklas Therning and Lars Bengtsson. Jalapeno: decentralized grid computing using peer-to-peer technology. In Nader Bagherzadeh, Mateo Valero, and Alex Ramírez, editors, *Conf. Computing Frontiers*, pages 59–65. ACM, 2005.

[123] Jan van den Bos and Chris Laffra. PROCOL — A parallel object language with protocols. In *ACM SIGPLAN Notices, Proceedings OOPSLA '89*, volume 24, pages 95–102, October 1989.

[124] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. 33(3):78–85, March 2000.

[125] W. Vanderperren, D. Suvee, M. A. Cibran, and B. De Fraine. Stateful aspects in JAsCo. In *Proc. of SC'05*, LNCS 3628. Springer Verlag, April 2005.

[126] Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component substitutability via equivalencies of component-interaction automata. *Electron. Notes Theor. Comput. Sci.*, 182:39–55, 2007.

[127] Kevin Viggers and Rob Walker. An implementation of declarative event patterns. Technical report, University of Calgary; Computer Science; Science, December 20 2004.

[128] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159 – 169. ACM Press, 2004.

[129] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.

[130] T. Ylonen. SSH - secure login connections over the internet. Proceedings of the 6th Security Symposium) (USENIX Association: Berkeley, CA):37, 1996.