Number of ordre: 2014-ISAL-??? Year 2014

THESIS

SECURITY AND SELF-HEALABILITY ENFORCEMENT OF DYNAMIC COMPONENTS IN A SERVICE-ORIENTED SYSTEM

defend at
L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

for the degree of
DOCTOR OF PHILOSOPHY

Ecole doctorale : INFORMATIQUE ET MATHÉMATIQUES

submitted at 28 February 2014

By
Yufang DAN

Defend at 14 May 2014 before the commission of exam
JURY

Directeurs	Stéphane Frénot	Professeur	INSA de Lyon
	Nicolas Stouls	Docteur	INSA de Lyon
Rapporteurs	Lydie du Bousquet	Professeur	Université Joseph Fourier
	Gael Thomas	HDR	Université Pierre et Marie Curie
Examinateurs	Ioannis Parissis	Professeur	Université Pierre-Mendés-France
	Frédéric Dadeau	Docteur	Université de Franche-Comté

This thesis is prepared at Centre d'Innovation en Télécommunications et Intégration de Services (CITI),
INSA de Lyon - INRIA Rhône-Alpes

Acknowledgement

First and the foremost, i would like to express my sincere gratitude to my advisors Prof. Stephane Frenot and Dr. Nicolas Stouls for their support and trust. They are most responsible for helping me complete this thesis as well as the challenging research that lies behind it. Their wide knowledge and their logical way of thinking have been of great value for me. They were always there to meet and talk about my ideas, to proofread and mark up my papers, and to ask me good questions to help me think through my problems. Without their encouragement, constant guidance, all their tolerance and patience, i could not have finished this thesis. I would like to give special thanks to Dr. Nicolas Stouls who has spent very much precious time to discuss my work and to help me analyse the existed problems and so on during the three years and a half.

I take this opportunity to thank the jury members for spending their precious time to read and review this thesis, for taking a long trip to attend this defence, and for giving me valuable comments. I am grateful for the financial support of the CSC through UT-INSA project.

I would also like to thank my dear colleagues at CITI laboratory for kindly, professional sharing the experience on their study and work, i had a very happy time in CITI for three years and a half. I specially would like to thank administrative staffs at CITI for their instant support whenever I asked for one.

Next, I must thank my dear neighbour, Danielle ROCHE, you are really like my family and my best friend in French, you let me felt i live at here like in my country, even though my french sentences sometimes were not clear, you are always there support me for my everything.

The last but not the least, I would like to extend my special gratitude to my big family: my mother, father, sister, brother-in-law and my husband. Their

unconditional support has always helped me to bounce back whenever i felt low.

Abstract

Dynamic service-oriented architectures (D-SOA) focus on loosely coupled client-server interactions where both of them can appear and disappear at runtime. Our goal is to design monitoring systems for these architectures. Since classical monitoring systems are statically injected into the monitored services, they can not properly handle the runtime services' lifecycle. Moreover, when a service is substituted by a new one, other services may still use the old reference. This reference is kept in memory as a stale reference which induces some forbidden behaviors.

This thesis contributes to design a monitoring system with resilient dynamicity that monitors services usage and is able to deal with stale references usage. This goal is achieved in three steps.

Firstly, by considering the dynamicity of SOA systems in an open environment, we design a corresponding dynamic monitoring approach. We identify two key properties of the loosely coupled monitoring system: *dynamicity resilience*, i.e., after the unregistration of a service, its interface monitor and its current state are kept alive in memory and transferred to a new loaded service; *comprehensiveness*, i.e., the implementations of the monitored interface can't bypass the monitor observations.

Secondly, to avoid stale references usage, we propose a client-side safe service usage (SSU) layer to automatically handle them. If a used service disappears, then the SSU layer can either transparently substitute it or throw an exception to the client. This SSU layer is based on a transactional approach which aims to preserve the coherence of active services.

Thirdly, we propose to integrate both approaches into a new monitoring system (NewMS). The NewMS inherits the principles of both systems: *dynamicity resilience*, *comprehensiveness* and *fault tolerance*. It can dynamically monitor service usage and transparently handle stale references of dynamic SOA systems.

All the three propositions are implemented on OSGi-based platform. We develop a simple application that simulates an Airline Reservation system, which is monitored by our monitoring systems. We also develop various automata to handle the dynamicity of the Airline Reservation system in the NewMS. Our results demonstrate that the time cost of our monitoring systems is close to one of classical monitoring systems.

Résumé

Les architectures dynamiques orientées services (D-SOA) se concentrent sur les interactions client-serveur à couplage faible, où les deux peuvent apparaître et disparaître à l'exécution. Notre objectif est de concevoir des systèmes de surveillance pour ces architectures. Comme les systèmes de surveillance classiques sont statiquement injectés dans les services surveillés, ils ne peuvent pas gérer correctement le cycle de vie des services d'exécution. En outre, quand un service est remplacé par un autre service, d'autres services peuvent toujours utiliser l'ancienne référence. Cette référence vers un service absent, lorsqu'elle est gardée en mémoire, peut induire des comportements non désirés.

Cette thèse contribue à la conception d'un système de surveillance de l'utilisation des services, qui soit résistant à la dynamique de la plateforme et qui soit en mesure de faire face à l'utilisation des références obsolètes. Ce but est atteint en trois étapes.

Tout d'abord, en considérant le caractère dynamique des systèmes SOA dans un environnement ouvert, nous concevons une approche de monitoring résistant au la dynamique de la plateforme. Nous identifions deux propriétés clés du système de surveillance à couplage faible: résilience à la dynamicité, c'est-à-dire qu'un moniteur d'interface et son état sont maintenus en mémoire et transférés à un nouveau service lors de la disparition d'un service utilisé, et exhaustivité, c'est-à-dire qu'un service surveillé ne peut pas contourner les observations du moniteur.

Ensuite, pour éviter l'usage de références vers des services qui ne sont plus actifs, nous proposons un service de sécurité côté client (SSU Layer), qui permet de traiter ce problème de manière transparente. Si un service utilisé disparaît, la couche SSU peut soit substituer le service de manière transparente, soit lever une exception pour avertir explicitement le client. Cette couche SSU est basée sur une approche transactionnelle qui vise à préserver la cohérence des services actifs.

Enfin, nous proposons d'intégrer les deux approches dans un nouveau système de surveillance (NewMS). Les NewMS hérite des principes des deux systèmes précédents: la résilience à la dynamicité, l'exhaustivité et la tolérance aux fautes. Il peut dynamiquement surveiller l'utilisation de services et traiter les références obsolètes de manière transparente.

Ces trois propositions sont implémentées dans la plateforme OSGi. Nous avons développé une application simple qui simule un système de réservation de place, qui est monitoré par notre systèmes. Nous avons également proposé différentes spécifications pour ce système. Nos résultats démontrent que le coût d'observation de notre moniteur est proche du coût d'un monitor classique, ne prenant pas en compte les problématiques liées à la dynamique.

Contents

List of Figures

Acronyms

AOP	Aspect-Oriented Programming
API	Application Programming Interface
DSL	Domain-Specific Language
D-SOA	Dynamic Service-Oriented Architecture
EJB	Enterprise Jave Bean
FSM	Finite State Machine
IPOJO	Inject Plain Old Java Objects
IDS	Intrusion Detect Systems
JavaMOP	Java Monitoring-oriented Programming
JBI	Java Business Integration
JML	Java Modeling Language
JSON	JavaScript Object Notation
JSON-WSP	JavaScript Object Notation Web-Service Protocol
JVM	Java Virtual Machine
LTL	Linear temporal logic
NewMS	OSGiLarva-SSU++ Monitoring System
OSGi	Open Services Gateway initiative
OSGiLarva	a Monitoring system with dynamicity resilience
PTLTL	Past Time Linear Temporal Logic
PVS	Property Verification System
REST	Presentational State Transfer
SOA	Service-Oriented Architecture
SSU	Safe Service Usage
STM	Software Transactional Memory
TM	Transactional Memory
UDDI	Universal Description, Discovery and Integration
WSDL	Web Services Description Language
XML	Extensible Markup Language

1 Introduction

Service-Oriented Architecture (SOA) is a software design approach which enables to build complex architectures made of independent services linked together at runtime. Each provided service is viewed as a single execution process, registered in a repository and linked with clients at runtime to manage requests. Most of the time the client is bound to the service, until it decides to release the service link.

In a stateless model, the client has no memory from previous calls. And the previous context is send over the link at each method call. In a stateful model, both the link and context data are automatically maintained between the client and the service. In the stateful case the client is bound to the service and if the service stops, the client must also stop.

In a stateful communication, information is slotted between the client and the server. During this period, neither the client nor the server can be changed. Although it works fine for instant transaction that consider client and server as fixed or stable point, we consider that when context is changed regularly in a mobile environment for instance, both server and client may be changed during stateful communication.

In dynamic SOA(D-SOA), client and server work together to agree on communication and data exchange protocols. Most of the time, they "know" each other and work

in a closed-environment. When considering a dynamic SOA system, client and services working on a stateful basis in an open-environment should have more guarantee between them. Therefore, this thesis proposes an analysis of the kind of guarantee client should expect from services and services from clients in an open-environment based on D-SOA.

In the following sub-sections, we first give an overview of D-SOA in Section 1.1. Then, the technical challenges for guaranteeing clients behaviors as expected at runtime in dynamic SOA-based systems are presented in Section 1.2. We give an overview of the main contributions of this thesis in Section 1.3. Finally, in Section 1.4, the organization of this thesis is presented.

1.1 Dynamic Service-Oriented Architecture Overview

SOA is composed of a large number of autonomous and self-contained services. Each functionality of an application is a service [72, 94]. A service in SOA-based architecture comprises service interface, service implementation and service contract. Service interface exposes the abstract functionality of services. Service implementation provides underlying business logic and data for fulfilling the specified functionality in the service interface. Service contract specifies service' functionality, binding protocol type and constraints for client service; it is also standard-based and platform independent and stored in a service repository [62].

Dynamic SOA (D-SOA) architecture consists of dynamic and loosely coupled services. The services' life-cycles can be dynamically managed remotely at runtime because of the loosely relation between client and service. For instance, services may appear and disappear dynamically in a regular basis without affecting the other services' execution. D-SOA framework has some rules for informing the corresponding service about the changed service life-cycle state (start or stop) or helping client to find a more suitable service implementation than the current used one.

Since services are un-associated and loosely coupled, services' interaction enables the invoking-side service to request server some functionalities through a repository that exposes appropriate contracts. Subsequently, the invoking-side service is bound to the service and is allowed to invoke methods through service interface as long as its contract types match. Moreover, some services can be composed together for becoming a new service with different functionalities at runtime and arriving at a new granularity level.

Due to these characteristics (loosely coupled, reusable, re-compose with different granularity levels), SOA has attracted more and more attention of large-scale firms

and wide areas. For example we can cite: RFID system based on SOA [73], Radioactive waste management domain [29], Data Mining field [103], Bio-medical data management [96], Cloud computing [26]. There are also different approaches to implement SOA among of the two families: web services and other more local approaches such as OSGi [1]. This last one is the object of our study and will be deeply introduced in Section 2.1.

In this thesis we focus on OSGi framework. It is usually used in 24/7 systems, where the system is not restarted when a service appears or disappears. This framework is targeted to embedded systems such as cars, ADSL boxes, or network systems. In such systems, web services cannot be used either due to the lack of connectivity, network limited bandwidth, or for efficiency reasons.

1.2 Motivations

Service-oriented architecture (SOA) is focused on loosely coupled client-server through public interfaces. The client usually requests service access through a repository. Subsequently, the client is bound to the service and is allowed to invoke methods as long as the interface types match. In dynamic SOA, each service invocation must be considered as a complete context switch since potentially new services may appear and others disappear at runtime, even if these services are stateful. This dynamic activity should have as few consequences as possible at the client side.

From a dynamic SOA point of view, dealing with loose coupling and dynamic issues of services are a real challenge today. Firstly, binding a client to a service is a matter of interface matching because of services loose coupling, but, neither the client nor the service has any guarantee that the other part behaves as expected. Secondly, every system implementing dynamic SOA faces the problem of deprecated references caused by the services mobility. Since a deprecated service reference potentially leads to a "null pointer reference" or to a wrong result, it can result in a system crash.

The objective of this thesis is not only to identify whether the behaviors from client are authorized or not in a dynamic SOA system. It is also to enhance the fault-tolerant characteristic of dynamic SOA system while service disappearance. The last but not least, in this thesis, all services may be regarded as stateful services in this kind of system. For achieving these goals, we check two cases:

First, it's important to continuously ensure the clients authenticity and the validity of the activities carried out after interface matching for most systems. Each time a client makes a request to a server, a formally specified constraint can be checked to ensure that the client is authorized to perform that call. So, a runtime monitoring system

can be used to check such behaviors in D-SOA systems. There exists some traditional runtime monitoring approaches for checking the specific behaviors of client accesses to a service. These approaches involve static mapping and monitoring of services, but there is a constraint from these monitors when a service disappears or a new one dynamically appears, these monitors can't continue monitoring the new replaced service without restarting system. This thesis defines a runtime monitor with resilience to dynamicity and comprehensiveness for dynamic SOA. In the light of such an objective, we explore the possibility of continuously monitoring new services request from clients without system reboot.

Second, in consideration of the valid state of service references in a dynamic SOA system, dealing with services dynamicity is important. A service reference is a specified pointer of client obtained to use its service in this system. So, we can propose some client side tools to aid the running dynamic SOA system. When service disappears, its system can still work or throw an exception. We can use these tools to add some codes at client side for fetching a new service reference as soon as a new service is available to replace the disappeared one. The client needn't to restart after this service substitution and it also avoids stale reference usage.

1.3 Contributions

In this thesis, the main contributions are listed as follows:

A dynamic monitor approach for monitoring a dynamic SOA system at runtime in an open environment is proposed:

- This dynamic monitoring approach inserts monitors at the point of client-server binding rather than "statically" at compile-time or loading-time. This approach can make dynamic mappings from monitor to service or method during runtime even if services appearance or disappearance, since the monitor has the same life-cycle with the monitored service interface rather than service implementation;

- This kind of monitor can check behaviours of clients using services and the other behaviors related to this service cannot bypass the monitor observations;

- Property description of this monitor is a composite of interface side property (i.e., Class-Property) and client side property (i.e., Instance-Property). These properties of this monitor can respective check the behaviours of each client using the service through its monitored interface with each client ID. The interface side property is the entrance of the monitor;

- An implementation of this monitor is realized by OSGiLarva system which describes method call events as well as OSGiLarva framework events in the property description;
- The monitor system also can monitor a complex system with multiple service interfaces and check the atomicity use of services. These interfaces properties can be described in the context of "global" respectively. They are distinguished by their interfaces name.
- This monitor generates a record and outputs to users or managers at runtime who can take some necessary measures to the monitored software system at the time of a particular state reached.

A " safe service use" layer at client side is proposed for enhancing self-healing capability of service usage in a dynamic SOA system.

- This layer is aware of stale references. It takes two steps at runtime for clients to prevent the use of stale references without requiring clients re-start and without modifying external services: if there is an new service for replacing the disappeared service, automatic make a service substitution for clients, else send a stale reference exception to clients.
- This layer uses transaction approach to ensure service coherent using at runtime. When a disappeared service is being used, the execution block rolls back and reverts all parameters values related to executed methods in it.

Finally, another dynamic monitoring architecture is proposed, which integrates the proposed OSGiLarva system and the SSU layer. It's used to monitor the secure of services usage and avoid the use of stale references of a dynamic SOA system in an open environment.

This proposed monitoring architecture named NewMS compensates the lack of OSGiLarva system by three ways:

- It is aware of stale references usage and handle it by SSU layer.
- It allows to express more precisely the properties. For instance, it is possible to consider the processing procedure of stale references.
- We designed an algorithm to automatically translate any OSGiLarva property into NewMS property.

1.4 Organization of thesis

This thesis is structured in five main chapters:

In Chapter 2, we first introduce the background knowledge of this thesis like web service architecture, OSGiLarva framework and some traditional monitor approaches usually used AspectJ technology. Then we survey the existing monitoring systems and classify those monitoring systems into different categories according to the bindings styles to the monitored systems. Finally, we survey the used approaches which tried to enhance the self-healable of services in dynamic SOA systems and control stale references using at runtime.

In Chapter 3, a dynamic monitoring approach for monitoring services usage in dynamic SOA systems for open environment is proposed. In this chapter, we express the architecture model for a dynamic runtime verification tool and consider some dynamic primitives. In order to implement this dynamic monitoring approach, we select two systems: LogOs systems and Larva system. We adapt and integrate both systems together as a dynamic monitoring system to support OSGi's dynamicity, and we call this tool OSGiLarva system. Moreover, for monitoring the behaviours of each client with its ID using services through different service interfaces, we analyse the situations and propose an upgrade property description with some new rules based on Larva property description language. Finally, we make quantitative benchmark tests to compare the OSGiLarva with a closed tool Larva and compare the monitored system with/without OSGiLarva system, and then analyse their performance.

In Chapter 4, a safe service use layer at client-side is proposed to enhance fault-tolerant characteristics of services according to the service disappearance in dynamic SOA systems. Firstly, we select a fault-tolerant technology (proposed in Chapter 2) to make software systems being more fault tolerant. Secondly, we give two parts to analyse the theoretical contributions of the SSU layer: (i) give requirements and policies for single service with a safe OSGi service reference, for instance, automatically enable service substitution and replay a part of the last comment or throw an stale reference exception to clients after a service unregistered, (ii) generalizing requirements and policies to the invocation of multiple services, for instance, automatically enable service substitution and re-execute its transaction block when a stale reference is used. From these theoretical contributions, we implement a SSU layer tool in the context of the OSGi environment.

In Chapter 5, we propose another new dynamic monitoring architecture (i.e., NewMS) applied to monitor services usage without stale references in dynamic systems for open environment. Since this NewMS is composed by the OSGiLarva system and the SSU layer, it still inherits the main principles from both tools. We show the new cases

possible thanks to this integration and we express the more precise NewMS properties.

In Chapter 6, we summarize the main findings of this thesis, the conclusion that can be drawn and some possible extensions of the work covered in this thesis are discussed.

2

Background and state of the art

In this chapter, we first give a introduction of the thesis background about the approach of SOA. In the rest, we make a survey of state of the art of monitoring tools which monitor software at runtime and finally we present the fault-tolerant capabilities of services in D-SOA systems.

2.1 Background

Due to the loosely coupled, reusable and re-composition characteristics of services, SOA are attracting more and more attention of large-scale firms. By the way, several approaches implementing it appeared. Among of them, Web Service architecture and OSGi framework are well-known and meeting different markets' needs. Another technology, Aspect-oriented programming, is used by some related works to aid the runtime verification systems. In this section, we will give details about Web services and OSGi framework, then give the reasons why our research focuses on OSGi framework in this thesis. AspectJ technology will be introduced at the end of this section.

2.1.1 Web Services

Web Services is one of implementations of SOA. It is a piece of code available on the network with the properties to be self-described and self-contained. It supports interoperability among different machines with concrete business functions over a network.

As shown in Fig. 2.1, there are three roles in web service framework [71]: Service provider, Service repository and Service consumer. Service Provider provides service implementations to realize specified service interfaces. Different service providers can develop different implementations for a same service interface to support the rapid service upgrading. These service implementations are independent. A role of service repository (e.g., services integration and deployment) is to manage all services from service providers. Service consumer can send messages to find a service from Service repository. If the consumer obtains a reply about the requested service, it can bind with this requested service implementation from service provider and can use it. The concrete service implementation is transparent for service consumer.

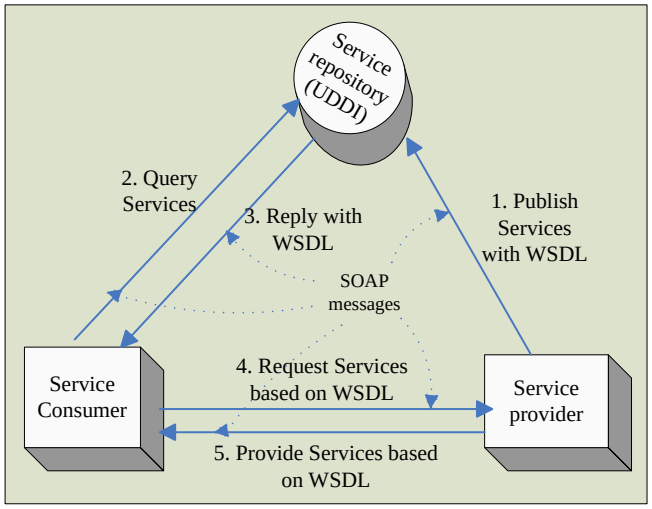


Figure 2.1: Roles and interactions in XML-Web service for implementing a SOA

There existed many markup languages used in web services designing, for example: JSON, JSON-WSP, REST and RESTful, XML-SOAP. JavaScript Object Notation(JSON) [97] which is a light-weight markup language for interchanging data on the web. JSON-WSP (JavaScript Object Notation Web-Service Protocol) [2] is a web

service protocol through JSON to describe, request and response services. Representational State Transfer (REST) [43] is a set of constraints and rules of architecture applied to the development of web services. It's also a resource-oriented architecture. A RESTful web API [81] is a web API implemented using HTTP and REST principles. XML-SOAP standard is exploited to describe, publish, find, match and configure web services [54, 101]. In this thesis, we propose to focus on the most used approach: XML-SOAP. We will then explain how to use it in Web services.

Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP) and services Universal Description, Discovery and Integration (UDDI) are three crucial platform elements in XML-SOAP Web Service architecture, which are also presented in Fig. 2.1. WSDL and SOAP are described based on XML format. WSDL [3] is used to describe web service information: transport type (e.g., SOAP), web service interface methods, parameters and web service URI. It is used to publish and request service. SOAP [4] is a service transfer mechanism in Web service architecture. It's used to exchange structure information of web service with other systems through HTTP. It avoids information conversion among different protocols. UDDI [40] is a registry center of services Universal Description, Discovery and Integration. It is used for registering new services with WSDL file through SOAP/HTTP protocols. It is like a yellow page of WSDL files. Service consumers can find registered services with WSDL files from UDDI through SOAP/HTTP protocol in heterogeneous and distributed environments.

Finally, Web services are taken as deals between enterprise internal and external, B2B and B2C businesses and so on. For example, in [29], a framework based on SOA concept and web service technology is proposed to manage a radioactive waste package record management system with three-tier. In [20], legacy systems' interactive functionality is exposed as web service by a wrapping approach to a system based on SOA. This solution made these legacy and heterogeneous systems become interconnected and interoperable over network. In [103], authors proposed a data mining service with data mining algorithms. This kind of service is taken as a web service for non-expert data miners in SOA. In [49], web services are designed based health care services in a SOA system. This kind of health care system can improve the quality of decision making and timely alert generation for doctors, caregivers and elderly people.

Except for Web Services, OSGi framework also is an important implementation approach of SOA. We will give its details in the following section 2.1.2.

2.1.2 OSGi Framework

OSGi services platform specification is created by the OSGi alliance in 1999. It defines a management model of a Java application life-cycle hosted in a virtual machine [9]. It has some APIs to manage the software components life-cycle from anywhere over network. The platform allows a remote loading and dynamic deployment of applications by its open specification in its environment while remaining independent of the system on which it is installed. OSGi services can run on different devices from very small to very big. Different service consumers, providers, developers, vendors can work well together on this platform specification. This framework implements a complete and dynamic component model based on a layered architecture.

This framework [8] consists of six main layers as shown in Fig. 2.2.

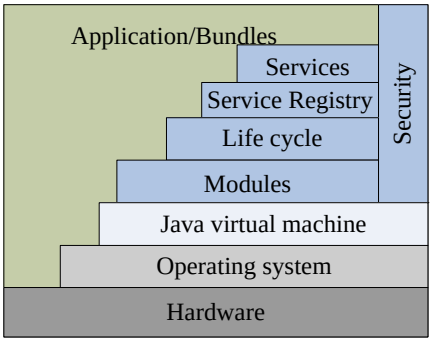


Figure 2.2: OSGi framework

1. Bundle layer: Bundle is the basic concept of the OSGi platform. It is the only unit of modularization and consist of a set of Java classes (packages, services), configuration files and other resources (e.g., images, sounds, etc.). This layer will work with all of the other layers. In each bundle, there are at least two methods: BundleActivator.start(BundleContext) and BundleActivator.stop(BundleContext). If the framework need to start this bundle, the former method have to be called. This method is used to register services or to assign any resources needed by this bundle. The later method is called when the framework need to stop this bundle. When this bundle is stopped, this bundle can't call any framework objects and it can not be called by any bundle until it started again.

2. Service layer: It offers a set of functionality for the publication, the discovery and the binding to Java objects, as well as the notification on the changes that occur on the services in the environment. A service is a normal Java object that is registered under one or more java interfaces by the service registry layer of

OSGi framework. The service is a solution is offered by the platform to avoid the tight binding between components. The binding can be done by using a service reference instead of a service object itself.

3. Service registry layer: This layer's API is used to manage services about *Service registration*, *Service tracker* and *Service Reference* generation. Bundle is used to register object by *Service registration*. The client services search the service registry with *service reference* to look for the matched objects. When a tool of *service tracker* is used in this service registry, it can listen the tracked service's *ServiceEvents* (e.g., Unregistering, Registered, Modified and Modified_ Endmatch) and obtaining and releasing service.

4. Life cycle layer: life cycle management for bundles provided by OSGi framework as some APIs can remote manage bundles start, stop, update, install and uninstall without requiring reboot. A bundle's normal life cycle is shown in Fig 2.3. The Starting and Stopping are middle states in a OSGi bundle life cycle. For example, when command "start" is executed, the bundle state is tranferred from "Resolved" to "Active". When command "stop" is executed, the bundle state is transferred from "Active" to "Resolved". It provides remote management for bundles with dynamicity.

5. Modules layer: A modularization module is defined for Java in this layer. The modularization module specifies encapsulation and declaration of dependency relationships among bundles: How a bundle can import and export code? What is the order among bundles export and import? [9].

6. Java virtual machine layer: It manages Java class-loading for multiple bundles. In local OSGi framework, multiple bundles run in a single JVM for sharing bundles and coordinating with other bundles.

From above introduction and consideration of each layers collection, we know that OSGi framework with its service registry provide a lightweight model to publish, find and bind services inside its JVM. This framework supports the characteristics of Service-oriented architecture. The life cycle layer provides APIs to bundles for managing services in module layer. These characteristics enable this framework to become a dynamic SOA approach. OSGi Service platform is being used widely: Home automation based on OSGi platform [63, 6, 100]; Vehicle industries adopted OSGi platform for supporting different vehicle manufacturers services. Moreover, it supports remote call vehicle service for unmanned vehicles [26, 78]; Desktop PCs, Servers (High-end Servers, including mainframe), Nokia and Motorola drove an OSGi technology standard for the next generation of smart phones [23, 57, 92].

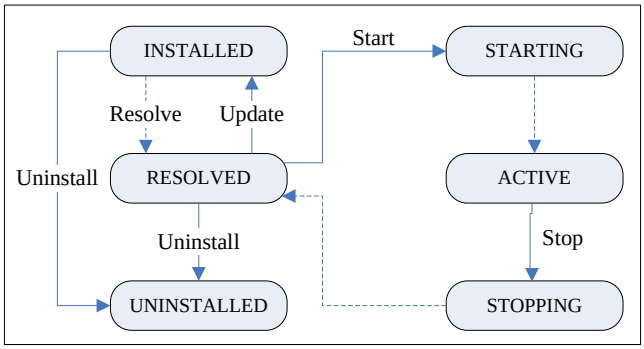


Figure 2.3: OSGi Bundle life cycle

Web service architecture is a popular implementation approach of SOA over network, while OSGi is a dynamic components model based for dynamic SOA. Here are main differences between web services architecture and OSGi framework:

1. Service view: Web services architecture would not typically be able to have the full view of the system, i.e., one can either observe the client or the server but not both. OSGi framework can reason about the full picture by also taking into consideration the OSGi framework events such as registration of services, service requests by different clients, etc. This is possible because OSGi framework provides remote and dynamic life-cycle management functionality.

2. Service messaging speed: There are different service transport mechanisms on both approaches. The local OSGi services communicate with each other just like general java invocations. All web services communicate with each other need to use SOAP binding with HTTP/TCP/UDP protocols. OSGi service methods are called at a thousands of times speed greater than the web service calls.

3. Service disappear at runtime: OSGi framework avoid the "null reference pointer" associated to the disappearance of a service without using "Service tracker". When a service has been loaded by a service consumer, this service consumer can invoke its service methods after it is unregistered. But for Web services, this invocation induces a null reference pointer at runtime.

4. Considering cost: All local OSGi bundles run in a single JVM for sharing and coordinating with the other bundles. This minimizes the memory footprint and improve performance. Because of this point, it provides almost zero cost among inter applications communication that is introduced in [8].

For the sake of these differences between Web services and OSGi approaches.

OSGi framework interested us to make a research. In this thesis, there are some things needing to pay more attention because of loosely coupled and dynamic treats [61, 8] in OSGi-based systems: dynamic monitoring services usage with validate service references.

At the end of background part, we introduce AspectJ technology because of some related monitoring tools using it (section 2.2).

2.1.3 AspectJ technology

There exists some close related works using AspectJ technology. Among of these, enforcement monitor [58], Larva [32], JavaMOP [75] and a dynamic monitoring system [102] are described in section 2.2. So, we briefly introduce AspectJ technology in the background.

Aspect-Oriented Programming(AOP) [60] is a programming paradigm. It uses *cross-cutting* approach to get common behaviors from the internal of packaged objects, and then encapsulate the common behaviors into a reusable model which is named *aspect*. The common behaviors affect multiple classes and are different with the business processes of objects, such as authority identification, Logging, transaction process and so on. The aspect enables the repeat code decreased, lighten the coupling among of modules and enhance operability and maintainability in a software system.

AOP implementations have some aspect expressions that can encapsulate cross-cutting concerns for software systems. AspectJ [10, 55] which is the most universally used AOP language is a seamless aspect-oriented extension to the Java programming language. It has some expressions to encapsulate the cross-cutting concerns into an aspect, such as *joint point*, *pointcut*, *advice*, *inter-type decleration*. The *joint point* is a class method from an original system. It is a abstract concept in AOP, it doesn't need to be defined. The *point cut* is a structure to capture the specified set of *joint points*. It just creates a link to the target system for observing. The *advice* specifies the execution code of point cut. It can give concrete execution logic with some special handling: *before*, *after* and *around*. The defined *point cut* will be executed *before* or *after* or *around* the captured joint point (e.g., class methods). The *inter-type declaration* is applied to declare the cross-cutting classes and their hierarchies. Therefore, the *pointcut* and *advice* dynamically handle the program flow at runtime, the *inter-type declaration* is done at developing-time. The aspect encapsulates these aspect expressions to form a clear modularization of crosscutting concerns. The aspect can be separated from target system and reused, such as error checking, monitoring, logging and so on [45].

This section 2.1 expresses the background of this thesis. It introduces the implementation approaches of SOA and makes a comparation to each other. Finally, AspectJ

is used by main of our related works have been presented in Section 2.2. The following two sections are state of the arts of this thesis: related monitoring systems(section 2.2) and self-healable of services in dynamic SOA systems (section 2.3).

2.2 Monitoring systems

In this section, we will give a state of the art about a variety of monitors for verifying static or dynamic systems' security of services usage.

Once service interface matched, it's difficult to guarantee safe usage of components in D-SOA systems. If we use a classical monitoring tool to check and verify some sensitive behaviors at D-SOA system runtime, service disappearance or appearance will induce undesirable things for the classical monitoring tool, such as, information lost, monitor disappearance and so on. Hence, two characteristics that we are thinking important in a monitoring tool for verifying D-SOA system: *resilience to dynamicity* and *monitoring comprehensiveness*.

- *resilience to dynamicity*: it refers to the preservation of the behavior flow. In case the monitored service is substituted, the monitor and its state should be transferred, meaning that the monitored property cannot be hard-linked to the code.
- *monitoring comprehensiveness*: it means that we cannot allow services to restrict what is observable by the monitor. If we want to check a property, we need to ensure that all the relevant events are monitored.

We propose to classify existing runtime verification approaches according to the monitor configuration with respect to the monitored software systems. The monitored property may be: manually written inside the code (in section 2.2.2), automatically injected inside the code(in section 2.2.3), kept out of the code (in section 2.2.4) and monitoring of web service (in section **??**). For analyzing *resilience to dynamicity* and *monitoring comprehensiveness* into each of these families, firstly we should give an explanation about some property description styles.

2.2.1 Properties classifications

Property expressiveness is an important characteristic of runtime verification systems. In this section, we will give a short properties classification: *Invariant property*, *Behavioral property*, *Liveness property*, *Timed property*. We will explain these properties on the following simple example:

```
public class A{
  public boolean aCallable;
  int x=1;

  public void m(){
      aCallable=true;
      x++;
  }

  public void g(){
      aCallable=false;
      x--;
  }

  public static void main(String[] args){
     aCallable=false;
     m();
     g();
  }
}
```

1. *Invariant property*: it's a property on state variables that have to be true *every time*. Depending on the observation granularity, it can be used between instructions or during each call of method. For instance, let us consider x, which is a variable of Class A. If we need to check that x is always larger than 0 during this class running, we need to define an invariant property for this class. This property can be expressed in this example by some data-oriented property description languages such as annotation by $/*@invariant\ x > 0;*/$, or such as an automaton (Fig. 2.4).

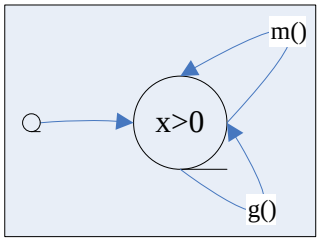


Figure 2.4: Invariant property described by an automaton

2. *Behavioral property*: It specifies some acceptable sequences of method invocations. For instance, after m() invocation there is always a g() during this example running, i.e., $m() \Rightarrow \bigcirc g()$. Such kind of sequence is taken as a behavioral property description in monitoring system. Finite-state machine(FSM) is the most usual way to describe a behavioral property. It also could be explained by annotation, but annotations are usually dedicated to invariant properties. If they are used to express a behavioral property, all related methods must be annotated. For example, by inserting $/*\ @requires\ aCallable == false; */$ as a precondition of each method, except for method g(). If any method is missed, there is no guarantee for the real executed results.

3. *Liveness property*: It is a more global case of behavioral property. It can not only specify some fixed sequences of method invocations, but also can specify some authorized sequences on infinite traces.

   For example, if m() is called, then g() will be necessarily called in the future. This property is expressed on an infinite traces. However, in the case of monitoring system, it is usual to consider a bounded liveness trace like that there is a limit of m() execution times before calling g(). Hence, this trace is a live property expression. If the states are fixed, we can use FSM to describe this sequence. Since the execution times of m() is not fixed in this liveness sequence (m() can be called 1 time or 2 times or n times before calling g(), all these sequences are correct for this liveness sequence), we can't express all these situations in one FSM. If we use linear temporal logic(LTL) to express this sequence, i.e., $\Box\,(m()\ U\ g())$.

4. *Timed property*: it's a behavioral property with time. For instance, after less than 10 seconds of m() call there is a g() invocation. The usual way to describe timed properties is a timed automaton. This automaton can express time constraints inside its conditions.

These forms of property descriptions will be used in the following monitoring systems discussions in order to describe acceptable behaviors.

2.2.2 Hard-coding

In this category, where properties are manually added at source code at developing time, we can cite all annotation techniques, like JML (in section 2.2.2.1) and Spec# (in section 2.2.2.2). In both cases, the monitor is not *resilient to dynamic* code loading. Indeed, if a part of the monitored system is substituted, then its monitor is removed, since it is in-line. However, this approach is interested in the term of *comprehensiveness*,

since we can observe anything in the program. A limitation of this approach is the dispersion of the monitor throughout the code, requiring significant intervention to write the property or to check that its description is correct. We give some details about JML and Spec# in the following sections:

2.2.2.1 Java Modeling language(JML)

JML [66, 67, 65, 19] is a specification language for a detailed design of Java modules. This modeling language is inserted into the java comments of java file. This language form is like the following annotations:

```
 //@ <JML specification>
or
/*@ <JML specification> @*/
```

When Java comment starts with sign @, this Java comment is translated as JML annotation. JML [19] is used to describe the behaviors of classes and methods from which users and/or developers can get the expected functions.

```
//@ requires descript != null;
 public String deleteAtAfterNl(String descript)
  { /* ... */ }
```

From above codes, we know that "requires" is a JML keyword. It's meaning a precondition can be defined before the method "deleteAtAfterNl(...)". Before invoke this method, system need to verify whether the variable "descript" is empty or not. If the "descript" value is non-null, this method can be executed. In the other case, a JML exception is thrown by system at runtime.

Since JML annotations are located in Java comment, they can't impact the compiling codes. When users and/or developers want to compare the actual behavioral from classes with the JML specifications, the open source JML compiler can be adopted. If the compared results do not match, the JML exception is thrown during the running of the java code. Some tools have been built around JML for unit testing [28], runtime checking [27], light-weight contract checking [22] and system verification [44].

For Dynamic SOA-based systems, these JML annotations are added in it at developing-time. When a service substituted by a new one, the monitor in the old one won't appear in the new one under the situation of without reboot. This specification hasn't *dynamicity resilience* for monitoring like Dynamic SOA-based systems.

2.2.2.2 Spec# Programming system

Spec# programming system [15, 16] is a new way to produce high-quality software by focusing on more cost effective. It consists of C# and Spec# annotation, compiler and static program verifier. The Spec# static program verifier is called Boogie. It can generates logical verification properties from Spec# program. These logical verification properties are added into the monitored source code level for static verifying. It focuses on three fields:

- to check *non-null types* from source code;
- to add *pre/post-condition* and *exception management* in method contract to verify its methods;
- to create *expose* block in class contract for constraining the data field of object invariants and class invariants.

JML and Spec# can express invariant properties which are methods granularity. However, some assertions can be added between each instruction for more deeply and targeted describing property. By the way, it is possible to encode behavior properties into them. But it can induce to add precondition before all methods. Spec# and JML are two close languages to verify original system. Since the Spec# directly adds its contracts (e.g., non-null type annotation, class and method contract) into C# code rather than into C# comments, it has larger design space with its specified contracts than JML to check and test systems. A more complete analyze about these two languages is given in [16, Section 3: Related work].

2.2.3 Soft-coding

In this category, where properties are injected at compilation time, or load-time, we can cite Enforcement monitor (in section 2.2.3.1), JavaMOP (in section 2.2.3.2) and Larva (in section 2.2.3.3). These tools use standalone description of a property and inject the monitors inside the code by AspectJ technology (in section 2.1.3), but this is not the same kind of hard-coding (in section 2.2.2).

Advantages of Soft-Coding approach are then the same as in the previous case, but specifying the monitor is easier, since the description of the property is centralized. However, these approaches from Enforcement monitor [58], Larva [32], JavaMOP [75] or a monitor dynamically inserted into OSGi service implementations by AspectJ technology at runtime [18] are *comprehensiveness* and only partially *resilient to dynamicity*; at best, the tool may inject the property at first-time binding, but once injected, the

property is hard-coded within the service for the whole execution of the class. We will give some details about these soft-coding monitoring approaches in the following.

2.2.3.1 Enforcement Monitor

In [58], the monitor is a proxy between a client and a server with the goal of checking time properties. In order to have sufficient time to check whether an observed timed property is correct or not, the runtime enforcement monitor focused on adding a fixed delay between the reception and the forward of input events. This property is sent by users to the monitor under the form of events sequence with delays which need to be delayed. If the primitives send to the active monitor and the delays are not the same with the given timed property, the enforcer will modify the delays by itself. It aims to make the output timed sequence conforms to the designed property. After that, the output timed sequence is taken as the input events sequence is sent to target system.

This enforcement monitor is explicitly called by client. It uses Aspectj compiler to weave these designed primitives into target system [86, 38]. However, it can not guarantee that all called method are checked through this monitor. Some methods can also be sub-called by other methods itself, not always by client. In such a way, the sub-calls are not observed by the monitor. It does not generate any input event and can not be considered in the enforcement monitor.

This monitor system can express behavioral properties and timed properties. The granularity of its property description is external methods.

2.2.3.2 JavaMOP

Java monitoring-oriented programming (JavaMOP) [24, 25] analyse framework dedicated to the monitoring of Java programmings, which accepts some independent specification formalisms. It aims at reducing the gap between formal specification and implementation by integrating them into its original system. It can be used to design a runtime monitor for developing reliability, security, dependability software. It can be used to design events' logics (e.g., FSM, PTLTL, LTL and so on.) in formal specification against software implementations. The designed specification is compiled by JavaMOP as AspectJ code, and then is woven into the target implementation system by any AspectJ compiler (such as ajc) [59].

The property description of JavaMOP can express behavioral properties and liveness properties by LTL and PTLTL. However, the LTL expressed live property will be generated by aspectj technology. The generated monitoring properties consists of several FSM formulas. There is no expresiveness gain in the monitoring property description, except easier to write it.

2.2.3.3 Larva Tool

Larva [32] is a tool which injects the monitoring code in a Java program to check a described property in a Larva script file. This tool which weaves calls interception using aspect-oriented programming techniques is closed to JavaMOP. Both of them permit to monitor some behavioral properties, but real-time properties could be expressed only in Larva. By the way, it can not only describe concrete service methods, but also control certain dynamic events occurring by timers.

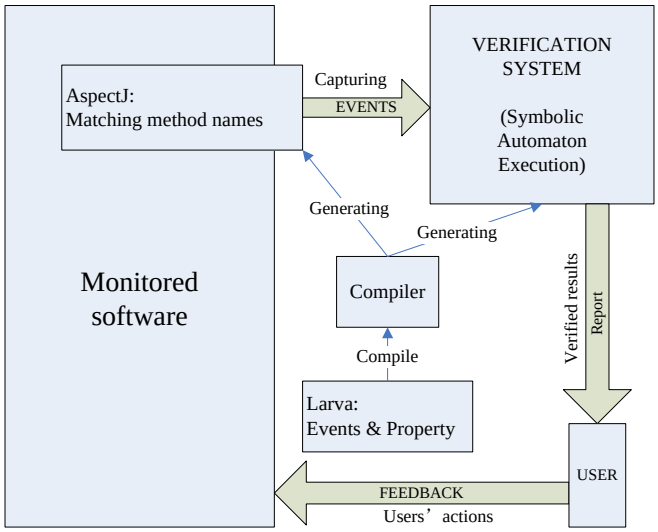


Figure 2.5: Larva system in a software system

When a monitored software is launched with Larva system(Fig. 2.5), its property script is compiled by *Larva compiler*. The *Larva Compiler* generates two main outputs from its script:

1. Aspect-oriented code: This code which links the monitoring code with the monitored software that aims to extract the monitored events. It will be statically injected some calls to the monitored software by using the *AspectJ compiler* at coding-time or at compiling-time or loading-time.

2. Java class code: This code is used to verify the extracted events conform to the designed property. The verification system is outside the monitored software. Once the designed event is checked, the verification system send the monitored records to users. It is up to the users to make some necessary actions to the target system when a monitored record is outputted.

Larva property description can express behavioral properties and timed properties. Its granularity is on methods(internal and external). Larva and JavaMOP have

really close characteristics in them. However, Larva performed better with regard to resources consumed than JavaMOP [33].

2.2.3.4 Monitoring of web services

There are a number of works (e.g., [102, 84, 14, 13]) that support the monitoring of web services. In [102], a dynamic monitoring framework with its monitoring scenario model and instrumentation layer is proposed for runtime monitoring SOA Execution Environment-based systems. In this approach, AOP instrumentation is used. Each exposed service has an interceptor socket code injected in, and wraps it with a socket. Each interceptor is taken as a service and is published with its interest and priority. Once an interceptor is registered, this registration information will be informed to every interceptor socket with its wrapped service for comparing interceptor's attributes with socket's. If attributes matched, this interceptor is added in the queue of matching sockets by its priority. Then the injected monitor can start to monitor the corresponded service invocation. There also exists some disadvantages:

- Interceptor socket code need to be injected to each exposed service, a socket is wrapped with this service, even injected into some completely needn't to be monitored services.

- This monitor currently just focuses on service invocation rather than specific invocation parameters or the implemented business logic. For example, it can monitor invocation rate and error rate.

The monitoring tool [102] can mainly expresses invariant property. And its property is instruction granularity.

Java Business Integration(JBI) is a kind of Web services model. Since AOP [60] can deal with crosscutting the aspects of a system's behavior as separately as possible and without forcing source code modification, an enrichment of JBI-compliant monitoring is implemented through AspectJ technology [84]. As shown in Fig. 2.6, the defined AspectJ pointcuts can be allowed to crosscut the JBI interfaces. Since keeping source code and class files avoid modification, the authors leverage load-time weaving these defined aspects by a dedicated java agent. This monitoring instrumentation based on AOP enrich JBI specification. Hence, this monitor can be a considerable restriction in the expression of security policies. AspectJ technology can be used to monitoring program points by its advice(be restricted to these manners: before, after, around), not the business processes logic. The enrichment of JBI-compliant monitoring can express invariant property. Its property is instruction granularity.

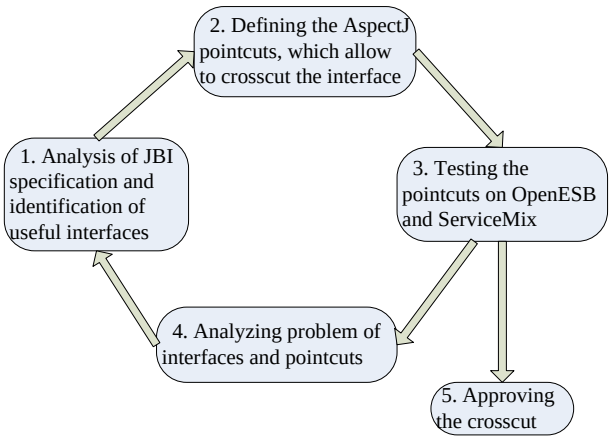


Figure 2.6: AOP crosscut analysis approach

In [14, 13], the authors provided both dynamicity resilience and comprehensiveness characteristics (even if these are not explicitly identified as such) by listening to events from a web service composition engine. Furthermore, in [13], this monitoring architecture supports both *instance monitors* and *class monitors*. The *instance monitors* check the behaviors of a single instance of BPEL business process; the *class monitors* extract or collect information from the checked behaviors of all instance monitors, it aims to get synthesized information at class point of view. However, to the best of our knowledge, no similar monitoring techniques have been proposed for the OSGi framework. Moreover, the context is not the same, since in a web service context, we can easily distinguish between callers by their IP address and port number, but it is impossible to know who is the caller, or which class or software is making the call. The monitor tool can express behavior properties, liveness properties and timed properties. Its property is business processes logic granularity.

Indeed, while it is technically possible to use AspectJ to support dynamic class loading and unloading in OSGi, then the monitored bundle must declare the import of the AspectJ library inside its Manifest file — an operation which is not really transparent to the service. Note that this restriction does not exists in Equinoxe implementation of OSGi (Eclipse). Since some choices would have been done in the configuration of the framework, requiring to restart the whole framework each time a new service is installed. Furthermore, if monitors need to be started or stopped at runtime it cannot be done directly through AspectJ without restarting the service—something which is undesirable in 24/7 services.

2.2.4 Agnostic-coding

In this third category, where the monitor is kept out of the code, we include any trace analyzes approach, such as LogOs system (in section 2.2.4.2) for monitoring OSGi-based systems, event log-based detection systems and logging systems (in section 2.2.4.1). The main advantage of these approaches is the loose linking between the property and the monitored system. Hence, if a package is substituted, the monitor can observe it inside the logs and the monitored properties are still the same for the whole system. Moreover, the description of the property is located into a single location, which facilitates property management.

However, such Agnostic-Coding systems can be bypassed, e.g., intrusion detection systems and logging systems can only observe what services accept to push. If a package provides a service without writing sufficient logs, then the monitor does not have sufficient information to check a event correlation [79]. LogOs system is better than that both monitoring systems, but we will see that some restrictions remain. In the following, we will explain each logging system.

2.2.4.1 Logging system

In [12], authors proposed an approach to UNIX security logging. This proposition used light-weight logging to off-line detect intrusion systems. They derive some empirical data from realistic intrusion experiments, and then use the derived data to compare with the light-weight logging methods which are a few simple methods. It aims to detect and trace attacks against original systems, for instance, it can be merged into an intrusion detect systems (IDS) [88]. But there is a precondition, enough data should be logged to make a better detection the intrusions or others. The event log-based detection system [87] also depends on sufficient empirical event logs of system. And its analysis depends on 3 years recorded event logs. Therefore, if logged information is not sufficient, this will affect the detections of logging system and event log-based detection system.

These logging systems can express behavior properties, liveness properties and timed properties. It is on methods (internal and external) granularity.

2.2.4.2 LogOs system

LogOs system [47] is a special logging tool based on the OSGi framework, developed at the CITI Lab during the LISE project [64]. It's designed to work in a dynamic SOA context. It can capture all behaviors of invoked service method during runtime, under the condition that the service's interface is marked by annotation, even its service

implementation is dynamical unloaded at runtime. As soon as the LogOs bundle is started, each service registration is observed. LogOs is a transparent logging toolkit for the service activity inside the OSGi architecture. Thanks to the OSGi hooking mechanism, a LogOs proxy is generated for each registered service. Hence, every method call from the annotated service interface, including parameters and returned values, are automatically intercepted. For each event captured by a LogOs proxy, a corresponding LogOs event-description is forged and propagated to LogOs. The event-description is just the service method who is annotated in the corresponding service interface. Then, the LogOs will record the trace and store it. LogOs system can intercept those specified service methods/parameters based on OSGi framework and store the logged events.

Since LogOs system's annotations add in specified service interfaces and its proxy adds between client and service implementation, LogOs system is separated from service implementation. When service implementation dynamic changed at runtime, the observation mechanism of LogOs system remain unaffected and no communication with this specified service interface can bypass the added proxy. So, this LogOs system has treats of both *resilient to dynamicity* and *comprehensiveness*. But in LogOs system, there is no verification part to check whether these captured behaviors from the running system are authorized or not.

LogOs system doesn't specify concrete behavior properties or liveness properties, it just give a constraint range to observe. It can observe all action between client using the annotated services. Its granularity is on external methods.

Finally, in this section 2.2, we gave the background and the state of the art on the monitoring systems part of this thesis. We know that a dynamic monitoring system with *resilience to dynamicity* and *monitoring comprehensiveness* is very important for supporting D-SOA systems. When services dynamic unload or substituted by other services, the special monitoring tool can restart at the latest event from the old one to continue monitoring the new one. The dynamicity of D-SOA system doesn't affect the dynamic monitoring system's observation mechanism and properities monitoring.

In the following, we discuss the second part of our state of the art: Self healable software systems.

2.3 Self Healab Systems

One of our contributions is to deal with dynamic issues of services in dynamic SOA-based execution environment. Hence, in this section, we will talk about state of the art of self healable software system. Some approaches and some related techniques(e.g.,

Fault tolerant technology) are proposed by some authors. Every model implementing this dynamic SOA-based system faces the problem of deprecated references caused by the services mobility. The OSGi component framework is one of the several models implementing SOA and in which stale references have harmful effects. In order to avoid the harmful effects according to stale references, the fault tolerant approach may be an useful solution and enhance the self-healable of services. Next we briefly introduce the occurrence of the stale references and dynamicity management of services in OSGi framework, then give the related works which have tried to resolve this issue.

2.3.1 Fault tolerant technology

A fault tolerant system [69, 51] consists in handling faults in order to system continue working and getting correct results. Fault tolerant computing can be taken as a system's capability to handle seamlessly error identification and recovery system from fault state to correct state [51]. It is commonly accepted that a general fault tolerance system has to go through four stages [95]:

- error detection
- fault location
- reconfiguration
- recovery and continued service

The error detection and fault location is meaning to use monitoring mechanisms for checking errors and locating faults. Currently there are many tools to deal with these two stages, for example, Logging analysis [85, 83], runtime monitoring approaches [36, 82]. The reconfiguration and recovery aims to redesign a correct execution plan for avoiding the located fault. There are usually three families of treatment to reconfigure and recover an error [37]:

- to mask the error;
- to roll-forward in the execution until a new stable state is reached;
- to roll-back to the previous stable state and restart the execution from it.

Usually, the mask an error mechanism consists in having redundant information. There are some redundancy techniques proposed for healing the checked system [50, 77, 80]. In [77], a self-assembly system was presented as having the potential to bring

about self-repair and regeneration: agents can replicate to substitute dead neighbors and thus even recreate the entire structure that may be lost. The Recovery-oriented computing research project [80] at UC Berkeley is also employing the fault tolerant technology to achieve the isolation of faulty components and provide redundancy techniques for fault safe online recovery.

For the roll-forward mechanism, the fault tolerance system will continue its execution until a stable state happened. Then the fault tolerant system handles its faults and the running system continues the execution. Finally, the roll-back mechanism is meaning that: before a fault occurring, the fault tolerance system stores all the required actions to be able to go back to a stable state after a fault occurring. This system can be re-executed with correct configurations for self-healing.

2.3.2 Self Healable systems in D-SOA

An existing approach for self-healable system is to use a D-SOA framework and to consider service substitution as a roll-back mechanism [37]. When considering the problem of services substitution, a complementary objective consists in achieving the substitution without any modification of the client code. Classical solutions are mainly considering this problem and can be grouped in three main categories [39]: abstraction-based approach, adapter-based approach and hybrid solutions combining the first two. The idea behind abstraction-based approach is to define higher level abstractions that stands for concrete services, and the client applications access to the alternative concrete service instead of access directly to the provided service. On the contrary, in the adapter-based approach, the client applications access directly to the concrete service through an adapter. Finally, the hybrid approach can reduce the complexity of the service number increased.

Whatever the approach used, the first part of the substitution is focused on finding a new service that can be used in place of the unavailable one. In [76], the authors propose an algorithm and mismatch trees to find incompatibilities in interfaces level and protocols level respectively. Checking the compatibility between all available services can be time-consuming, and at the same time impact the ongoing business process. Some approaches reducing this complexity have been proposed. The main idea is to gather available services into groups of services offering the same functionalities [46] [39] [89] each client application is bounded not only to one service, but to a group of services. In SIROCO [46] framework using semantic annotations in SA-WSDL language in order to categorize services into OWL ontology. In [39], the authors propose to group available services for the substitution in groups called profiles. In [89], these

authors proposed a common interface(namely Open Service Connectivity OSC) to collect web services and dynamic binding of web services. In this work, web services substitution has two steps: collects functionally-similar web services into communities and makes client applications connect to web services communities using OSC driver.

After finding the right service, substitution can now be realized that is the second part. To realize a substitution is to reconfigure the system so that client services could keep working by using the new service. In [17], authors propose an algorithm for CORBA service reconfiguration, that involves a passive link to the unavailable service and an active link to the new service, while keeping the application consistency and with a few execution disruption. In the case of stateless services, it is straightforward. But for stateful services, it is more complex. One should restore the state of the substituted service. In SIROCO [46] framework, there is a registry system, where a service can register its current internal state and thus make a checkpoint. When a service fail, the framework try to manage the new service in order to set its internal state in the late one of the previous service. A synchronization mechanism has been presented in [98]. The configuration manager provides a runtime kernel which provides a message repository for messages that has been sent by components.

Almost all the aforementioned approaches are server-side and do not tackle stateful services. For stateful services substitution, one should implement a transaction mechanism to restore the state of the substituted service. Transactional memory provides more powerful support for this *lock-free* style of programming. Massalin and Pu [74] use this instruction for lock-free list manipulation in an operating system kernel. In [56], transactional memory is introduced with a support of a multiprocessor architecture to make lock-free synchronization. Lock-free data structures can avoid common problems: Priority inversion, Convoying, Deadlock. This method performs better than the locking-based data structures. Verification the correctness of transactional is also an important step during the transaction memory. So, Cohen et al [30] provided a mechanical proof of the soundness of the verification method and studied safety properties in situation where transactional code has to interact with non-transaction memory accesses. In [53], authors present the first approach to verify STMs under relaxed memory models with atomicity of 32 bit loads and stores, and read-modify-write operations. They use FOIL to automatically check the correctness of STMs under this model. [70] proposed an algorithm tracks object visibility at runtime by multiple threads are automatically guarded by transactions. Programmer allowed to use TM and needn't to explicitly manage whether objects are accessed transactionally or not.

In this section, we presented the classical approaches having tried to do service dynamic substitutions with stateful or state-less services at sever-side. It makes services

become more autonomic in D-SOA systems. Since, service providers can make any assumptions on provided services with the objective that service substitution can be done without any consequence on the client side, service references may become stale during clients using after service unloaded. This performance issue caused by the service dynamicity of D-SOA systems. So, we explain the stale references of OSGi-based systems and its resolutions in section 2.3.3 and section 2.3.4.

2.3.3 Stale references in OSGi

The OSGi platform allows a remote loading and dynamic deployment of applications. A service is a running java implementation, whose interface is available in an open repository and using a reference of service instead of a service object itself. But, this reference also has a drawback: the referenced service can be stopped and its dependencies deprecated at the moment of its use, leading to a stale reference. A stale reference is a reference to a service that is no longer available, either because of the bundle offering that service has been stopped or the service associated has been unregistered [9]. Client bundles may not be aware of the disappearance of the service or service references are deprecated at runtime. We are focusing on the case of a mobile platform with OSGi that can discover or lose connection to some service providers. In such a case, a service requested by a client can be lost while in use.

Writing safe code for handling OSGi service references boils down to properly listening to the OSGi service registry and tracking which services are in, and which services are out. This also requires that each call to a service in a client code makes extra steps. That is effectively going to invoke a method on a service whose reference is not staled. This is not easy as it seems, as concurrency is involved. Indeed, a thread may be invoking a service while another one is un-registering it. This easily defeats guarded accesses to a service reference if no intrinsic locks or fine-grained re-entrant read/write locks are being used.

To illustrate this, let us consider a class that is part of the core OSGI API: *org.osgi.util.tracker.ServiceTracker*. Briefly, this class handles the service appearance and disappearance tracking logic based on a set of service interfaces and filters. It can be used to fetch one or multiple service references at a given instant. It is widely recommended to use it when dealing with the OSGi service layer. Nevertheless, it does not handle concurrency and multi-thread OSGi bundles may use stale references or throw exceptions when taking advantage of it. The following piece of code, part of a demo OSGi bundle activator, throws a *java.lang.NullPointerException* because guarded access to a service reference is not correct in this concurrent setting. Moreover, when calling two times the service, you can get two different services, which can generate errors in case of stateful

services.

```
public void start(BundleContext bc) throws Exception {
  publisher = new Thread() {
    public void run() {
      while(true) {
        ServiceRegistration registration =
          bc.registerService(
            HelloService.class.getName(),
            new HelloServiceImpl(),
            null);
        registration.unregister();
      }
    }
  };
  publisher.start();

  final ServiceTracker st = new ServiceTracker(
     bc,
     HelloService.class.getName(),
     null);
  tracker.open();

  invoker = new Thread() {
    public void run() {
      while (true) {
        if (st.getService() != null) {
          ((HelloService)st.getService()).hello("World!");
          ((HelloService)st.getService()).hello("Second!");
        }
      }
    }
  };
  invoker.start();
}
```

Indeed, the *publisher* thread continuously publishes and removes a service, while the *invoker* thread continuously invokes it using the indirection of a service tracker. A race condition causes the *NullPointerException* to be thrown.

The following piece of code is the same one, but the got service reference is stored in memory. The guarded access to a service reference may become staled during both calls in this concurrent setting. But it guarantees that the used service in both calls is the same one.

```
public void start(BundleContext bc) throws Exception {
  publisher = new Thread() {
    public void run() {
      while(true) {
        ServiceRegistration registration =
          bc.registerService(
            HelloService.class.getName(),
            new HelloServiceImpl(),
            null);
        registration.unregister();
      }
    }
  };
  publisher.start();

  ServiceReference sr = bc.getServiceReference(HelloService
     .class.getName());

  invoker = new Thread() {
    public void run() {
      while (true) {
        HelloService hs=(HelloService)(bc.getService(sr));
        if (hs != null) {
          hs.hello("World!");
          hs.hello("Second!");
        }
      }
    }
  };
  invoker.start();
}
```

From this code block, we know that the *publisher* thread continuously publishes and removes a service too, while the *invoker* thread continuously invokes it. A race

condition causes the current used service reference becoming *stale reference*. The *hello()* method is invoked at the second time through the reference of the unregistered service.

This observation stresses out the fact that OSGi service references need to be manipulated carefully: it is easy to run into race conditions when multiple threads execute running concurrently, and it is easy to perform a method invocation on a reference that throw a *NullPointerException* by using service tracker or became *stale*.

2.3.4 Dealing with Dynamicity in OSGi

When a bundle becomes unavailable, all the references to objects it provided should be released to allow garbage collector to do its work correctly. In [48], we have an example of a case in which the substitution process fails because of a mishandling of stale references. The stale references should then be tracked and destroyed. Many approaches have been developed to detect and when possible delete these stale references. OSGi specification released some advises to use ServiceFactory Interface or Indirection mechanism in service object implementation in order to limit the consequences of stale references. In [48], by using Aspect Oriented Programming techniques, the authors propose a tracking stale references tool named Service Coroner that helps to find stale references for developed or maintained OSGi applications, and apply it in two cases study. Others approaches such as using Service Binder [21] or IPOJO [42] suggest to separate functional and non-functional aspects, by describing the services dependencies management information in meta data XML files and merge the both at the run time. Each of these approaches tackles a particular case of the stale references problem, but a general solution is not yet provided. An alternative solution is the use of a proxy [7], instead of a service references. The proxy manages load-/unload of services and the client services do not longer keep a reference to a likely disappeared service and the problem of stale reference is then avoided.

2.4 Summary

From above all, we introduced the background of this work and we explained our reasons to focus on OSGi framework. However, in order to monitor the communications of services usage without stale references in D-SOA systems, we try to use classical monitoring systems to monitoring it. We listed some related works of classical monitoring systems in the first part of state of the art. They have their advantages and disadvantages. But they are not enough to satisfy the dynamicity of services in D-SOA systems. In addition, the fault tolerant technology may be an useful approach to

enable services' self-healable in dynamic SOA-based system ease development. We introduced some classical solutions to deal with the stateful/stateless services dynamic substitution in D-SOA-based systems without any consequence on the client. It makes services more autonomic and dynamicity. But the stale references and the null pointer exception will occur at runtime in D-SOA systems because of service dynamicity, for instance, in OSGi-based systems. Since client doesn't know whether these things happened or not, they can lead to incorrect results or even system crash. There are some approaches tried to handle the dynamicity in OSGi. But it's not complete to solve these issues.

Therefore, we will give our propositions to monitoring the communications of services usage and enhance the fault tolerance of dynamic SOA-based systems in chapter 3 and chapter 4 respectively. Chapter 5 is a final contribution merging the first two systems.

3

A Monitoring Framework for Supporting Services' Dynamicity

Service-Oriented Architecture is an approach where software systems are designed in terms of a composition of services. OSGi is a Service-Oriented Framework dedicated to 24/7 Java systems. In this Service-Oriented Programming approach, software is composed of services that may dynamically appear or disappear. In such a case, classical monitoring approaches with statically injected monitors into services cannot be used. In this chapter, we propose a dynamic monitoring approach dedicated to local SOA systems, focusing particularly on OSGi. Firstly, we define two key properties of loosely coupled monitoring systems: *dynamicity resilience* and *comprehensiveness*. Next, we propose the OSGiLarva tool, which is a implementation targeted at the OSGi framework. Finally, we present some quantitative results showing that a dynamic monitor based on dynamic proxies and another based on aspect-oriented programming have equivalent performances. These propositions were presented in [35, 34].

3.1 Introduction

As stated in Chapter 1, we know that services are loosely coupled and client invokes service methods as long as this service interface matched in SOA-based system. Monitoring a critical system based on D-SOA is a challenge. Many runtime monitoring tools exist, but their properties are injected into the monitored system at coding time (JML [19] and Spec# [15]) or at loading time (enforcement monitor [58], Larva [32] and JavaMOP [75]). It means that when a monitored service based on D-SOA system disappears or replaced during runtime after bindings, its monitored property is also removed from system if the monitored system is not restarted. Moreover, these works don't consider the expression of properties in terms of framework events.

In this chapter, our proposal is to bring a dynamic approach to runtime monitoring systems through inserting monitors at the point of client-server binding rather than "statically" at compiler-time or loading-time. This means that both the service bindings and the behavioural monitoring bindings are dynamic and loosely coupled, thus supporting service substitution. This approach would preserve behavioural monitoring states across different service versions and check that both versions are behaviourally compatible.

Another major concern in a highly dynamic context, where the implementation of an interface may be substituted, is to ensure that no implementation, or part thereof, can bypass the monitoring framework. Note that if this could happen, the monitor would not be able to detect any malicious code which might be executed. Moreover, what can be concluded about a system's observation if some events could have been missed? Our aim is to enable the monitoring system to be fully active, even if the service provider ignores it.

In this context, we conjecture that a dynamic runtime monitor must have two significant traits: *dynamicity resilience* and *comprehensiveness* which are introduced in section 2.2 and reminded in section 3.3.1. Note that we are not assuming that every service behaves as expected, but only that if an authorized service is to be checked for a particular property, then no event of the service behaviour can bypass the monitor observations. For this reason, the architecture relies on a generic event-interception mechanism and a dynamic, loosely coupled, wiring mechanism for automaton verification.

The contribution of this chapter is a generic approach as well as a tool based on OSGi. In this tool, the verification logic of the automaton is handled by an adaptation of the existing monitoring tool Larva [32]. Finally, the introduction of dynamicity to the monitor also increases the scope of properties we are able to address. Thus,

we introduce some dynamic primitives in the property description language in order to make it possible to describe behavioral properties, where the registration/unregistration of a service are expressible events. Furthermore, we also adapt the life cycle of properties, since, under different circumstances, the monitor state might need to be preserved or reset when the underlying service is substituted.

Section 3.2 is a case study showing some requirements of this proposition. Section 3.3 expresses the architectural model for a dynamic runtime verification tool and takes into considering some dynamic primitives. Section 3.3 introduces our OSGi reference implementation and describes our modifications of the Larva specification language in order to consider dynamicity. We also analyse the property discription with different interface numbers suit for OSGiLarva. Section 3.5 illustrates the OSGiLarva tool by some quantitative results. Finally, Section 3.6 shows our initial conclusions.

3.2 Example

In order to ease the understanding of our contribution, we give an example of a dynamically monitored system conforming to our proposition. Let us consider an embedded client on a mobile device based on a dynamic SOA platform, which needs to communicate with a distant system according to a particular protocol Fig. 3.1. Let two services S_1 and S_2 provide an identical interface to access the distant system through different media: S_1 using a WiFi connection, and S_2 using a 3G connection. With such a configuration, we can consider that each time the WiFi connection goes down, the system unregisters S_1, effectively switching the client onto S_2, and vice-versa.

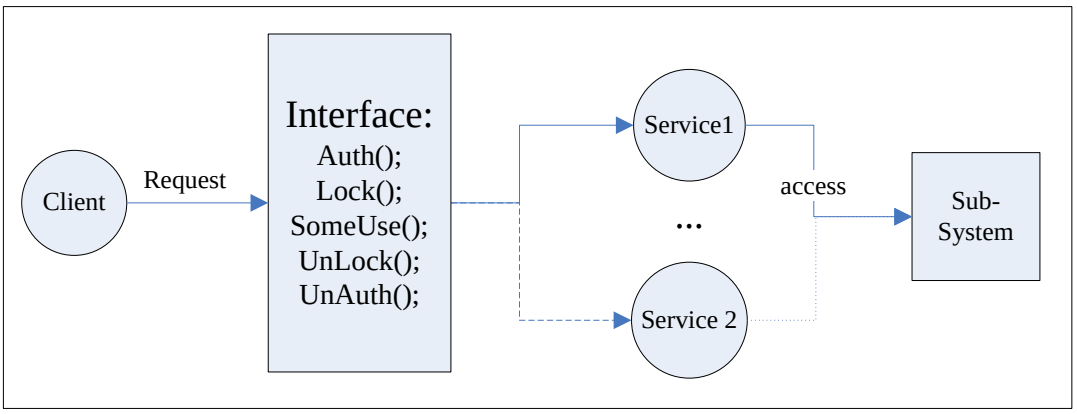


Figure 3.1: Dynamic SOA system supporting service substitution

Moreover, we consider that the use of the distant system requires that the client

is authenticated with the service and that some system actions have to execute atomically. Such requirements correspond to any typical secured system supporting concurrent access by transactions.

In such an example, the possibility of service substitution is crucial. We then propose, in Fig. 3.2, an example of an execution scenario that has to be supported by the system. In this scenario, the service S_1 is substituted by S_2 during the atomic part of the run.

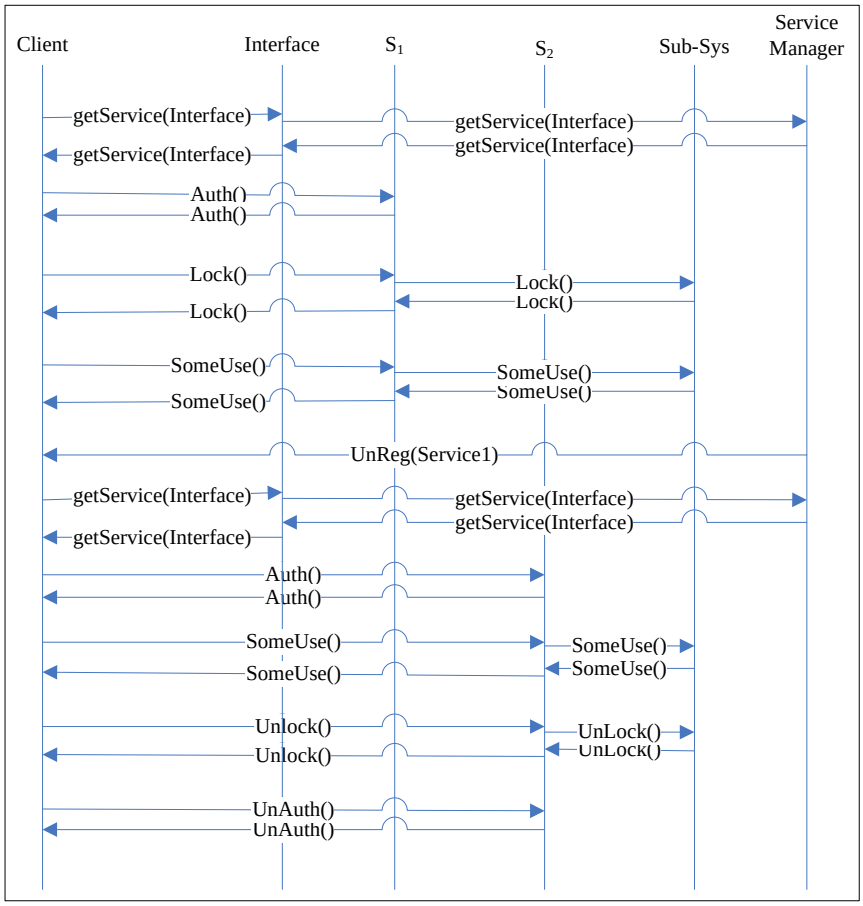
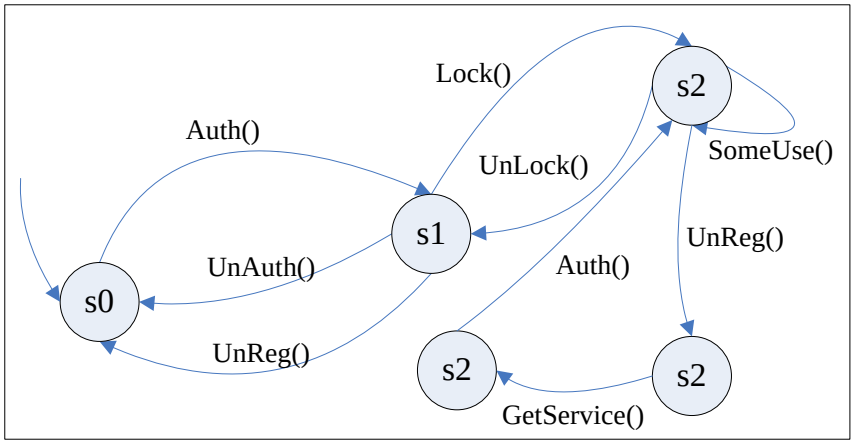


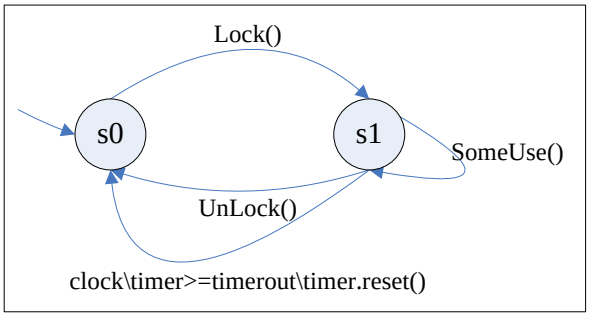
Figure 3.2: Example of scenario with dynamically monitored system supported by example in Fig. 3.1

In another part, we can describe the correct use of the system in some property and check it by monitoring at runtime. For instance, the two following properties express the expected behavior, described earlier: (i) the client is locally authenticated on the service before using it, and (ii) the concrete use of the sub-system requires that the client opens the lock and closes it after use. In this example, one would like to ensure that the execution described in Fig. 3.2 is correct with respect to these properties. Such

verification and the description of the property itself are the main contributions of this Chapter.



A. Client-side: instance property



B. Interface-side: class property

Figure 3.3: Example of a property associated to example in Fig. 3.1

These properties can be described by a couple of automatons (Fig. 3.3), but with a different interpretation of each. The local authentication automaton (Fig. 3.3.A) is maintained in case of service substitution and should be instantiated for each distinct client using the system. In the following, we will call such properties as *Instance-Properties* as they are instantiated on a per object basis; in this case a client. On the contrary, the management of the atomic use of the sub-system (Fig. 3.3.B) needs to be centralized and shared by all clients. Even if a service is removed and substituted, we would want to keep the current state of the sub-system in memory. In the following, we call such properties *Class-Properties* because its lifetime spans throughout the system's life cycle and is not bound to a particular entity.

In summary, our proposition is to provide a monitoring framework, which is able to monitor such properties by listening to method calls and OSGi framework events

in a dynamic, resilient, and comprehensive manner.

3.3 Contributions

In the first part of this section, we describe an abstract architecture of a monitoring system model supporting specific features of dynamic SOA systems, and we discuss its characteristics. In the second part, considering the dynamic primitives from the dynamic SOA system. At the end, we give a general property descripton in our monitor model for monitored system from three point of view: server side, client side, service interface side.

3.3.1 Proposition of a generic architecture

Our proposition consists in dynamically inserting a monitoring proxy in front of each service, and executing monitors in some autonomous services (Fig. 3.4). When a service usage event occurs, a notification is sent to each associated monitor, which checks the event against its property.

An interesting advantage of using a dynamic proxy over AspectJ, is that we can start or stop the monitoring of a property without restarting the service. Indeed, since the proxy is bound upon a service request, this can be handled easily, while AspectJ aspects are bound at least at class load-time, requiring to restart the service.

Since services are treated as black boxes from the running environment's point of view, such an architecture is designed to consider only properties of their external interface. This corresponds to properties expressing the normal authorized use of a service. However, since we are considering dynamic systems, we also want to consider dedicated framework events, such as unregistration of a service or getting a new service. In this approach, we will then focus on behavioral properties.

Since several clients can be running simultaneously within the framework, the scope of properties should not be restricted to the use of a single client. We consider the possibility of adding a monitor in front of several client. By considering both the monitoring of Instance-Properties and Class-Properties, we enable the possibility of simultaneously checking both local as well as global properties on the system.

In order to enable properties expressed in terms of method call events and framework events (requests, registration, unregistration, etc.), we need to capture both kinds of events — the ones between the client and the service, and some events from the service registration system. To inject a monitor between a service and a client using it, we adapt the framework in order to make this invisible both to the client and the service. Two interesting characteristics of this approach are that it does not change the

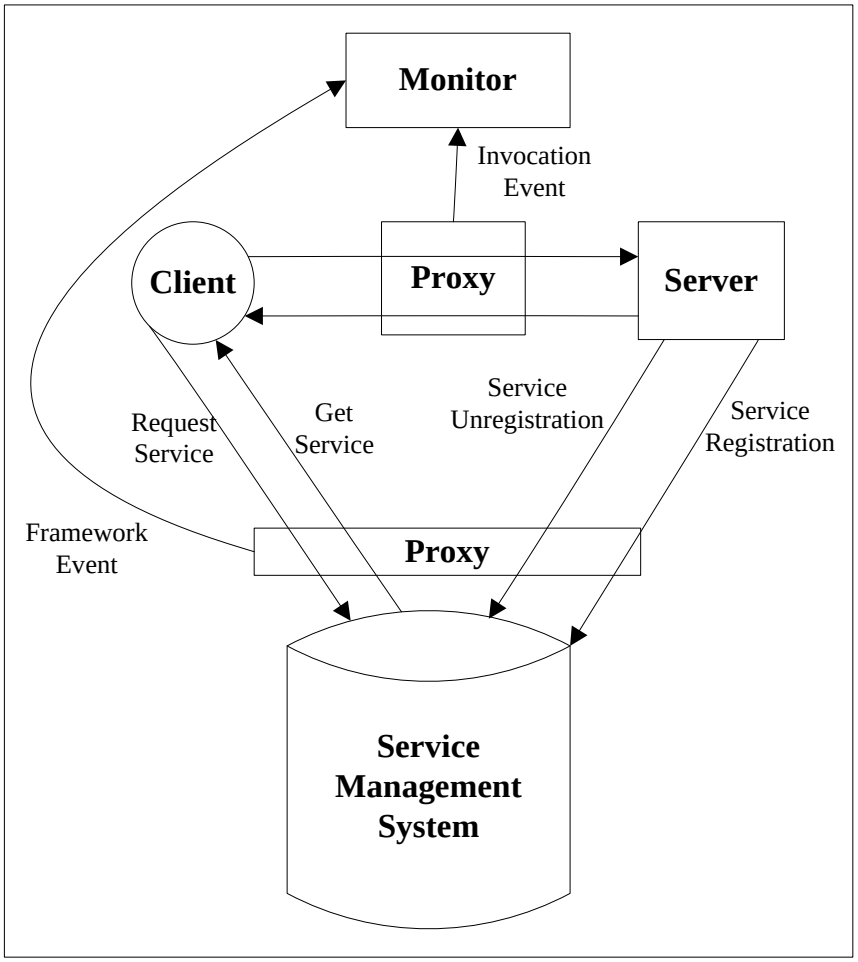


Figure 3.4: Proposed abstract architecture for monitoring system

binary signature of the service and that neither the service, nor the client, are aware of a potentially running monitor. By adding another proxy in front of the service management system of the framework, we are notified of requests for getting service references.

Fig. 3.4 describes the abstract architecture. In the following, we delve deeper into our two main principles.

Resilience to Dynamicity Since the monitoring system is externalized in an autonomous service, monitors are separated from the code. When changes occur in the framework, the observation mechanism and its properties remain unaffected.

Comprehensive Monitoring One of the main concepts of dynamic SOA is to have a framework which allows dynamic loading and unloading of loosely coupled services. Since the framework is in charge of providing an implementation to each service request, the framework can add a proxy between the client and the service to observe their communications. This observation is comprehensive and no communication can bypass this proxy.

3.3.2 Considering dynamic primitives

A monitor is started when a monitored service is registered in the framework. From this moment, each event related to this service (e.g., service method invoking, service loading etc.) is propagated to this monitor. Since we are in a dynamic framework, dynamic events can occur, e.g. un-registration an registered service, or loading a registered service. We propose to introduce the four following primitives:

- **REGISTER**: this event occurs when a new service implementation is registered on the framework. It means that a client can now get this service reference at any time. If another implementation is already registered, it shares the same interface property.
- **GETSERVICE**: this event occurs when a client is asking for a service. It can lead to two situations: client gets a service or the client does not get any service. If a client couldn't get a service from the server, it means that there is no registered service corresponding to the client request. We introduce the **NOGETSERVICE** event to handle this case.
- **UNGETSERVICE**: this event occurs when a client releases a loaded service. Each client can release its service object respectively and this service also exists in memory for other clients load and use.
- **UNREGISTER**: this event occurs when a service is unregistered. The created service object is then considered as destroyed. However, if clients still use this service, these actions are considered as perhaps no longer safe or functional.

3.3.3 General property description

This part discusses the property description language and focuses on the scope of the property description, mainly induced by the location of its associated monitor. Indeed, since we are not in a system with one client and one service, we could have many clients using many services at the same time. In such a case, the location of the

monitor can change the point of view of the property and hence its expressiveness. As general point of view for property description, it can be defined with at least three possibilities (e.g., Fig. 3.5): (i) client point of view, (ii) service implementation point of view and (iii) interface point of view.

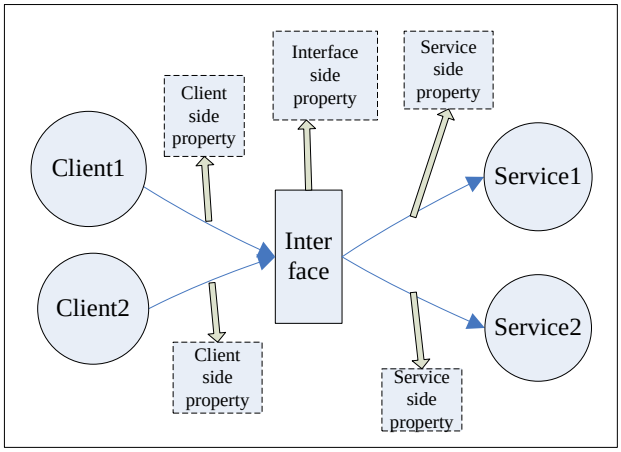


Figure 3.5: Possible point of view for properties

Next, we will give the details about the property description from the three points of view of monitored system:

3.3.3.1 Property Described from Service Side Point of View

If the designer describes a property with this point of view, shown in Fig. 3.6, he/she considers the use of a single service [99]. It is easy to consider some behavioral dependence in some parallel uses by multiple clients. However, since we are considering automaton-based properties, it is not obvious how to distinguish between clients within the property. Moreover, it is complex to consider the use of multiple implementations of an interface simultaneously, with potentially some communication between them.

For the dynamical part, it is not intuitive to describe and use the fact that a new implementation of the same service interface has been loaded on the platform. Moreover, it seems to be complex to share property memory between implementations of the same interface. Hence, if a service is substituted, there is no means of keeping its property in memory, with its internal state, and to map it on another implementation designated to continue the started work.

Advantages:

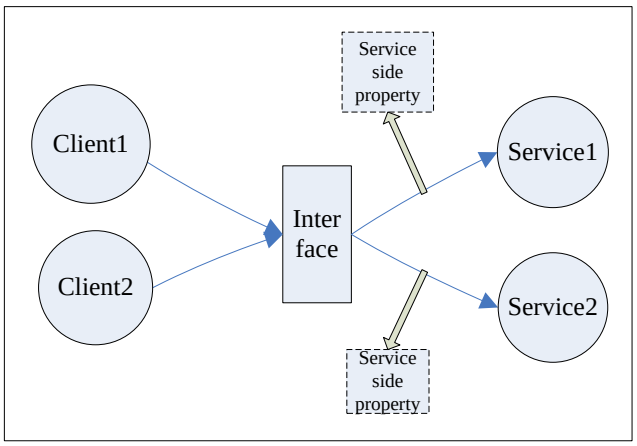


Figure 3.6: Property description: service implementation point of view

- Simplicity to describe behaviors of each service implementation without the need to make the link with other possible implementations.
- In case of stateful services, with a different memory address space for each implementation, it is very easy to describe the system.

Disadvantages:

- Complexity to describe shared memory between services.
- Impossibility to describe a generic behavior for each client, since we cannot distinguish between clients.

3.3.3.2 Property Described from Service Interface Point of View

In this point of view, we consider what can be done through a service interface, is showing in Fig. 3.7. It is easy to describe the global use of any implementation of this interface by any client, but not to make distinction between clients or between used implementations.

By its nature, such a property is not directly associated to a service and thus describes a property shared by all implementations. Note that it is easy to consider the loading or unloading of a service implementation, even if it is a substitution, willing to keep the current state of the property.

Since our property description language is automaton-based, the only manner to consider parallel use of many clients is to make some composition between the property and itself. However, such technique leads to a combinatorial explosion of the

automaton size. Moreover, it limits the maximum number of clients and services, since we need to have this information to make the composition.

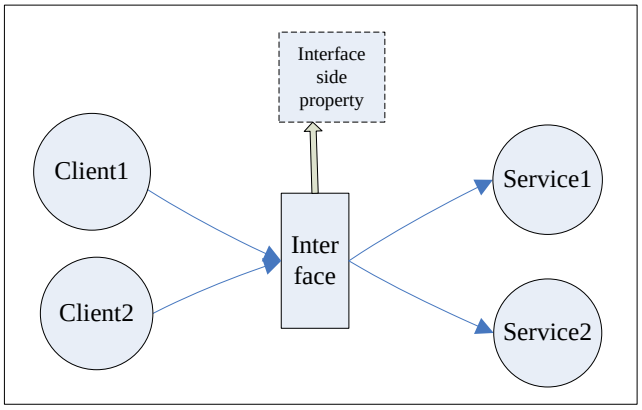


Figure 3.7: Property description: service interface point of view

Advantages :

- Easy to make a description of the authorized uses, with a global point of view
- Easy to consider loading/unloading of implementations
- Possibility to share a single property state between service implementations

Disadvantages :

- Risk of the shared property description size explosion if we want to describe the concurrent behaviour of several clients.
- Impossibility to describe a generic behavior for each client, since we cannot distinguish between clients

3.3.3.3 Property Described from Client Point of View

This third possibility considers that each client has its own instance of the property (Fig. 3.8). Hence, it is easy to describe the correct use of a service from one client point of view and to consider as many parallel uses as we want, without any combinatorial explosion.

Moreover, it is easy to describe the use of multiple services by a single client and the behavioral dependence in case of concurrent use of services. In case of substitution of a service, this approach can be resilient, since the property is attached to the client.

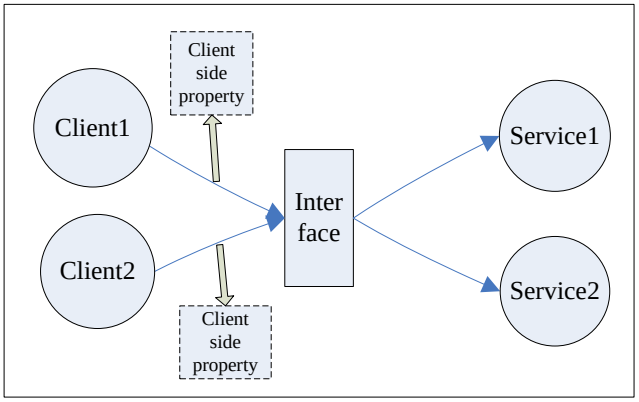


Figure 3.8: Property description: client point of view

However, in case of the simultaneous use of a single service by several clients, if there is some interactions between these usages, it is more complex to describe it.

Advantages :

- Easy to make a description of a particular client authorized usages
- Easy to consider loading/unloading of implementations
- Possibility to share a single property state between several service implementations
- No risk of size explosion of the shared property, since it cannot be described

Disadvantages :

- Complexity of describing global behavior including several clients

Hence, these three point of views are complementary. If each one of them is used alone to describe property for dynamic SOA systems, it's not enough. In section 3.4.1, we describe our choice of property in the light of the three types in the monitoring system for OSGi-based systems.

In the next section, we present our monitoring tool implementation based on OSGi framework.

3.4 OSGiLarva — A monitoring tool for OSGi

We propose a concrete implementation of the described monitor system model in the context of the OSGi framework: OSGiLarva (Fig. 3.9). In our tool, we use Java mech-

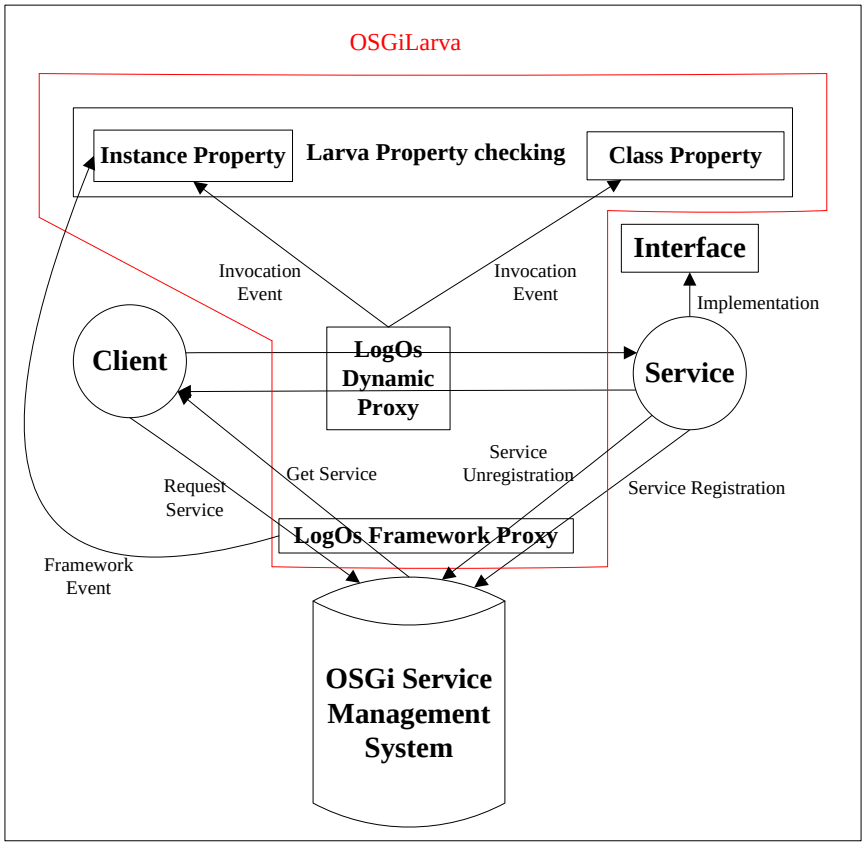


Figure 3.9: OSGiLarva implementation

anisms in order to generate a proxy between each client and service. This proxy is dynamically generated from a framework proxy, hooked onto the OSGi framework, and listens to all framework events such as the introduction of a new service or the requesting of a service by a client.

This implementation integrates two existing tools introduced in Chapter 2.2: Larva [32] and LogOs [47]. Larva tool belongs to soft-coding and LogOs system belongs to agnostic-coding. We use LogOs as a hooking mechanism to observe services' interactions. Larva is a compiler which generates a verification system expressed in Java language. We will use an adaptation of Larva to verify property events which are transferred by LogOs.

We describe the monitor implementation with following parts: We first introduce the property expressiveness with dynamic primitives of OSGiLarva system and then explain the OSGiLarva property description language. Next, we present our OSGi-Larva implementation with both LogOs and Larva systems. Finally, we describe how the registration process of a service under OSGi will take into account an existing

property monitor to insert it between the service consumer and the service itself.

3.4.1 Property description of OSGiLarva

The OSGiLarva description language is originally based on the Larva property description language. We adapted it in order to support more dynamicity. This adaptation is done through three extensions. The first one is the introduction of framework-event primitives (i.e., section 3.3.3) and a property as a composition of *Class-Properties* and *Instance-Properties* in the property description language. The second one expresses the syntax and semantics of OSGiLarva automata. The last one describes a complete OSGiLarva property in OSGiLarva property description language.

3.4.1.1 Using dynamic primitives in OSGiLarva system

Larva uses as input a property description language based on automata, extended by timers, variables and actions. In the property itself, the user defines the set of symbols used in the automata. These symbols are events which, in the original version of Larva, are defined in terms of method names. We thus propose to use the dynamic primitives, described in Chapter 3.3.2 in the events definition in order to enable framework-event : Register, UnRegister, GetService, NoGetService. UngetService isn't included among them, since it requires that the client uses some interfaces of the OSGi framework to capture UngetService operation. To use the UngetService and GetService of the interface `ServiceFactory` is a part of our future work. Currently, we just focus on the other events which corresponds to the event-descriptions generated by the adapted LogOs version. So LogOs needs to register some listeners on the framework.

Event **GETSERVICE** is obtained by using an OSGi `FindHook` instance, registered in the OSGi framework. When registered, such object is called each time a service is obtained. Originally, this mechanism was defined in order to make a filter on services obtained as a result of `getService` call. Indeed, the `getService` method accepts as an input a description of the expected service and returns an array of corresponding service implementations among the available ones. The `FindHook` mechanism has been introduced in order to allow service filtering (i.e., to hide some services). Note that LogOs also uses this mechanism to ensure that, if a service is monitored, every calls to this service are necessarily done through a proxy, and never directly.

REGISTER and **UNREGISTER** events are obtained by registering an OSGi `EventHook` with the service management system. An object implementing the `EventHook` class and registered in the framework is called each time the service management system

observes a modification, such as new incoming service, a service un-registration, or a service property modification.

In each of these cases, an event descriptor is forged by LogOs and sent to the Larva monitor. Larva treats such events like all other events. Hence, the event descriptor is compared to the list of events the monitor is listening to, and, if the property is expecting this kind of event, it triggers upon it.

In front of advantages and disadvantages of approaches to express properties (described in Section 3.3.3), we propose to consider properties as a combination of two kind of properties for OSGiLarva, associated to two point of views: client-side and interface-side. These two points of view in our monitor are respectively called **Instance-Property** and **Class-Property**. We propose to not consider the service point of view, since in typical use of OSGi, if multiple services implement a single interface, the framework favours the use of the same implementation by all clients. Moreover, from our experience, we conjecture that properties are typically client side, since an interface property cannot consider the concurrent use of services by many clients without a state explosion. Finally, to have the possibility to add a centralized property, interface properties can be useful to express some shared constraints such as locking/unlocking systems.

Since our contribution is based on the Larva description language [31], chosen for its closeness to our requirements, we mainly orient our proposition according to Larva and adapt it in order to support more dynamicity. In Larva, properties are described by automatons, where a single script file can contain several automatons. Moreover, Larva provides in its language the possibility of defining parametrized automatons which can be instantiated using event parameters, through the FOREACH keyword. We exploit this characteristics in order to use properties composed by two parts (Instance-Property and Class-Property):

- **Instance-Property:** If a property is defined as an Instance-Property, then each time a new client accesses the interface, a new instance of the property is generated and added inside the monitor. When the client terminates, the associated instance of the property can also be removed. Hence, while such properties are still resilient to service implementations' dynamicity, they are intentionally not resilient to clients' dynamicity. The framework events are useful to describe each client's fact state and behaviors.

- **Class-Property:** This case corresponds to a centralized property, meaning that several clients using a particular interface will share the same Class-Property. Such property is more resilient to dynamicity since a Class-Property can be kept

in memory until the associated interface is unloaded. As such it is not associated to a particular user's interaction or a particular service implementation, and can thus be used, for instance, to express some centralized locking/unlocking mechanisms. It's necessary to describe method calls in Class-Property. The framework events which will be described in Instance-Property for each accessed client are useless in Class-Property. However, if several implementations are used concurrently, then they would probably need to be synchronized.

Next, we present our adapted Larva property description language structure in the context of OSGiLarva for making more dynamicity.

3.4.1.2 OSGiLarva automata: syntax and semantics

In this section, we propose to formally define OSGiLarva properties in terms of *correct* and *bad* execution traces. An execution trace is error ending if and only if, it makes the property automaton reaching a bad state (defined in the property). An execution trace is correct, if all reached states of the automaton are only non-bad states.

Instance properties and Class properties are similar in there definition. They differ only by their life-cycle. We then firstly define what is a property with its syntax and semantics before to zoom in their particularities.

We define the structure of a property by an automaton, and then we define what means crossing a transition. Finally, we make the semantic association between a property and the set of its correct, or bad execution traces.

Definition 1 (OSGiLarva property automaton) *An OSGiLarva property automaton is a 8-tuple* $A = (S, s_0, B, V, v_0, \Sigma_M, \Sigma_F, \delta)$ *defined by:*

- S*: a finite set of states' names;*
- s_0*: the name of the initial state of the system* ($s_0 \in S$)*;*
- B*: a non-empty finite set of bad state names* ($B \subset S$)*;*
- V*: a set of variables names, defined in the property;*
- v_0*: a set of variables initializations of* A*;*
- Σ_M*: a finite set of events names, associated to call of methods;*
- Σ_F*: a finite set of events names, associated to framework events.* $\Sigma_F = \{Register, GetService, NoGetService, UnRegister\}$*;*

- *δ: a transition function defined by $\delta : S \rightarrow \Sigma \rightarrow [Prop_V \times Act_V \times S]$, where the first S characterizes the starting state, $\Sigma = \Sigma_M \cup \Sigma_F$, Σ is the set of all possible events (i.e., method calls or framework events), $Prop_V$ is the set of propositional conditions based on variables from V, Act_V is the set of actions based on variables from V and the last S characterizes the reached state.*

A transition t from δ is defined from a starting state to an ending state and with three annotating elements: an event $e \in \Sigma$ (method call or framework event) that triggers the transition, a condition, having to be true before to cross the transition, and an action, which is executed just after the trigger event occurring and the conditions of this transition are true. The action is some code that can manipulate property variables or generate lines in the record of the monitored OSGi system (i.e., Log file). Concretely, it is some Java code in the OSGiLarva property file. This code is restricted to not generate any event during its execution (for instance, an event of loading a new service).

Particular cases: if no transition can be triggered, then the automaton keep its state without doing anything. It doesn't correspond to a bad execution, but to an abstraction. Finally, if multiple transitions can be crossed (occurring event is their trigger event and their condition are true), then the transition that is firstly defined is the one that is crossed. This choice induces the use of a list in the definition of the transition function.

More formally, we can define the action of crossing a transition as following:

Definition 2 (Crossing a transition) *Let A a property automaton, $s \in S$ its current state and v the current valuation of its variables V. Let $tr = \delta(s)(e)$ the list of transitions of A starting from s and associated to a fired event e.*

We define a function $crossing(tr)$ and get a result that includes the reached state and the new variables valuation of V after the transition tr crossed in A through this function:

1. *If tr is empty, then s and v are unchanged:*
   $crossing([]) = (s, v)$

2. *Else, if the condition of the first transition tr_0 of the list is true under the valuation v, the transition is triggered and the action of tr_0 is executed on variables V. Else, if this condition is not checked, then the function is recursively called on the rest transitions $tail$ of its list tr:*

$$crossing(tr_0 \circ tail) = \begin{pmatrix} (verify(tr_0, v) \Rightarrow (reaching(tr_0), exec(tr_0, v))) \wedge \\ (\neg verify(tr_0, v) \Rightarrow crossing(tail)) \end{pmatrix}$$

where $tr_0 \circ tail$ represents the concatenation between the head and the tail of the transitions list, $verify(tr_0, v)$ is a function which is true if and only if the condition of tr_0 is true under the valuation v, $reaching(tr_0)$ is the state reached by the transition tr_0 and $exec(tr_0, v)$ is the valuation obtained after the execution of the action of tr_0 on the valuation v.

Using the first two definitions, we can define one step transition in a whole property automaton by the following:

Definition 3 (Stepping once in an automaton) *Let e an occurring event. Let A be a property automaton. Let tr_{A_e} the list of transitions of A starting from s and associated to the event e ($tr_{A_e} = \delta(s)(e)$). After the consumption of the event e, the automaton A is in the state s' and its variables are defined by the valuation v' defined by: $(s', v') = crossing(tr_{A_e})$*

We write $e \vdash A_{(s,v)}$ the stepping of e on A, returning $A_{(s',v')}$.

Finally, the call of a service method is atomic. Thus, the parallel execution of multiple clients using a single interface can be define as a single trace, interleaving events coming from different clients. The verification of a whole system according to a full property associated to an interface can then be defined as follows.

Definition 4 (Full property verification) *Let t_{c_i} the execution trace of a client c_i as a consumer of the monitored interface. Let $e_{(c_i,j)}$ the j^{th} event of t_{c_i}. The global execution trace t_{global} is then a mixed execution trace which interleaves traces of all clients using the given monitored interface.*

$$
\begin{aligned}
t_{c_0} &= e_{(c_0,0)}; e_{(c_0,1)}; e_{(c_0,2)}; \ldots; e_{(c_0,n_0)} \\
&\ldots \\
t_{c_m} &= e_{(c_m,0)}; e_{(c_m,1)}; e_{(c_m,2)}; \ldots; e_{(c_m,n_m)} \\
t_{glob} &= e_{(c_i,0)}; \ldots; e_{(c_j,n_j)}
\end{aligned}
$$

Let P a full OSGiLarva property, defined by a class-property (A_{class}) and an instance-property (A_{inst}). Then checking the use of the interface consists on checking $P = (A_{class}, A_{inst_0}, \ldots, A_{inst_n})$, with the global trace induced by the execution of current clients.

Let $inbad(A)$ a function returning true if and only if automaton A is in a bad state. Then we can functionally define the verification of a system execution according to a full property as follows, where the returned list is the sequence of clients reaching bad states (A.k.a. the log file). We write $check(t_{glob}, P)$ this verification process of t_{glob} according to P and we define it by:

- $check([], P_{A_{class}, A_{inst_0}, \ldots, A_{inst_i}, \ldots, A_{inst_j}}) = []$
- $check(e_{(c_i,j)} \circ tail, P_{A_{class}, A_{inst_0}, \ldots, A_{inst_i}, \ldots, A_{inst_j}}) =$
  $let\ A_{class(s_m', v_m')} = (e_{(c_i,j)} \vdash A_{class(s_m, v_m)})\ in$
  $let\ A_{inst_i(s_n', v_n'))} = (e_{(c_i,j)} \vdash A_{inst_i(s_n, v_n))}\ in$

$$if(inbad(A_{class(s_m{}',v_m{}')}))then[c_i]else[]$$
$$;if(inbad(A_{inst_i(s_n{}',v_n{}')}))then[c_i]else[]$$
$$;check(tail, P_{(A_{class(s_m{}',v_m{}')},A_{inst_0},...,A_{inst_i(s_n{}',v_n{}')},...,A_{inst_j})})$$

In this verification process $check(e_{(c_i,j)} \circ tail, P)$, $e_{(c_i,j)}$ is the first event of the global trace, $tail$ is the rest of the global trace without the first event of the global trace. The $check(tail, P)$ is used to make a recurrence for verifying the first event of $tail$ with the property $P_{(A_{class(s_m{}',v_m{}')},A_{\text{inst}_0},...,A_{inst_i(s_n{}',v_n{}')},...,A_{\text{inst}_{n_j}})}$.

After an OSGiLarva automaton formally presented by a 8-tuple A, these specifications are used to the next section of OSGiLarva properties description language.

3.4.1.3 Properties description language of OSGiLarva

In **Larva**, one way to introduce a new context is to use a **FOREACH** clause. This clause is a quantification on an object. Hence, for each instance of a given class, Larva generates a new instance of the inner property. Moreover, in order to make distinction between users, classical Larva needs information explicitly given by the caller, such as a Session ID, passed as a parameter. Hence, Larva only has the same information as the service implementation to check a property.

We propose to adapt the **FOREACH** structure for our needs, by introducing a new clause: **foreachclient**. This construct is based on the address of the caller and generates a new context for each caller. As an example, such a clause could make it possible to check that there is no ID session spoofing, what is not possible in larva. A described property in the **FOREACHCLIENT** context will be re-instantiated for each client using the monitored service. It is the instance-part of the property.

A very important difference between the **FOREACH** and **FOREACHCLIENT** clauses is that the first one is based on values computed inside the **EVENTS** clause from observed parameters, while the second one is based on values computed and provided by LogOs, according to its observations.

The **FOREACHCLIENT** keyword takes two parameters: *Long pid* and *String itfName*. *pid* is a numerical identificator of the client associated to the current instance of the property. Concretely, we use the process id as identification. *itfName* is the name of the service interface associated to the current event. The values of both parameters are directly transferred from LogOs observation. Since **FOREACHCLIENT** is an extension of the **FOREACH** clause, then we keep all language characteristics of the latter.

Besides **foreachclient** clause we need to explain **GLOBAL** clause. The property of each service interface is instantiated only once and is then shared by all clients. These kind of properties are *Classes-Properties*. These properties will be expressed in the **GLOBAL** clause.

It's possible to express a property in terms of several service interfaces. To make it, it is sufficient to write as many properties in the global clause as service interfaces to check. The link between the monitored interfaces and the properties is syntactically done by the name properties which must be the name of the interface. These properties share the variable definition and events definition in the context of **global**. It can communicate with all "instance-parts" of the property.

In order to make distinction among same name of methods which are provided by different interfaces, we design a format of valid event definition in each $EVENTS$ clause to express an exactly method invocation. For instance, $eventName = \{interfaceName \cdot methodName(type, ...)\}$, where $eventName$ is the defined event name, $interfaceName \cdot methodName(type, ...)$ is a method invocation and $type$ list are method's parameters types.

In the next section, we express the OSGiLarva property description and monitor given execution traces for an example system.

3.4.1.4 Verification example through OSGiLarva automaton

In this section, we give an example of Airline Reservation system with its property and we propose to monitor execution traces which are correct or not. This example (Fig. 3.10) is composed by two service interfaces: $reservation$ and $payment$. The $reservation$ interface provides two methods $selectFly(String)$ and $confirm(boolean)$ and is implemented by two services S_1 and S_2. The $payment$ interface has a single service implementation S_3, providing two methods $pay(String)$ and $resetPay()$.

Clients must use both services methods with a special order, such as, before paying, clients must select a fly, or before confirming, clients must pay. If the current used reservation service (i.e., S_1) is unregistered before confirming a client's airline reservation, the payment service need to be re-setted, and then client gets a new reservation service (i.e., S_2) from initial state to make an airline reservation.

We first give a part of a property file (shown in Fig. 3.11) which is composed by an Instance-Property (i.e., clients' property) and two Class-Property (i.e., one property per service interface) according to this example.

We then give an OSGiLarva property automaton definition of the *clients* property, which is the instance property of Fig 3.11. According to the **Definition 1**, a transition is defined under the form: $\delta(s, e) = [(c, a, s')]$, where s is a start state, e is an event, s' is a reached state, c is a condition and a is an action. Since this property automaton describes the instance-part property for each client, this property is described in a FOREACHCLIENT clause. The variable $itfName$ is the parameter of **foreachclient**. In this context, a property automaton of this example could be defined as following:

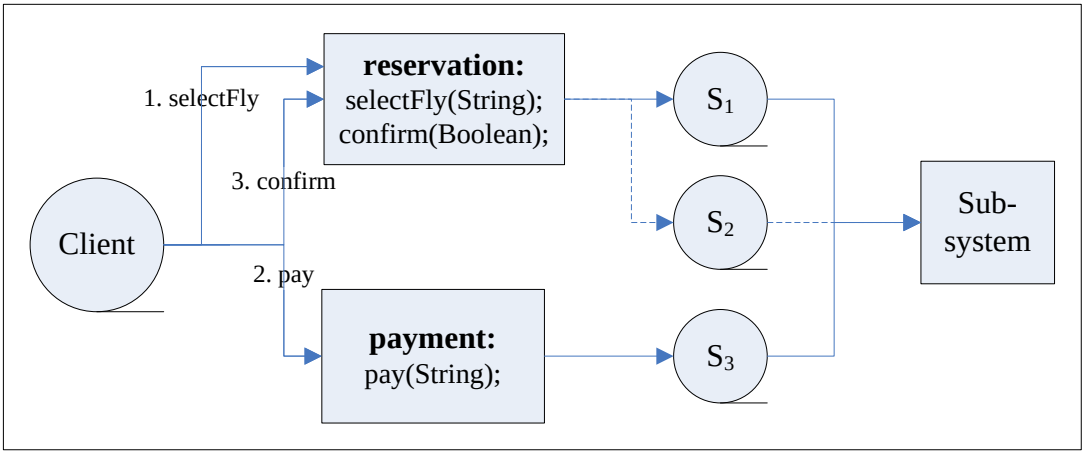


Figure 3.10: Monitoring of services usage

- S={$linked$, $selected$, $error$};
- s_0=$linked$;
- B= {$error$};
- V=$\emptyset$;
- v_0=$\emptyset$;
- Σ_M= {$selectFly, pay, confirm, resetPay$};
- Σ_F={$GetService, UnRegister$};
- $\delta = \{$
  $\delta(linked, selectFly) = [(true, nop, selected)]$,
  $\delta(selected, selectFly) = [(true, nop, selected)]$,
  $\delta(selected, confirm) = [(true, nop, error)]$,
  $\delta(selected, pay) = [(true, nop, paid)]$,
  $\delta(paid, confirm) = [(true, nop, linked]$,
  $\delta(Unreg3, resetPay) = [(true, nop, selected)]$,

  $\delta(selected, UnRegister) = [(\text{"}reservation\text{"}.equals(itfName), nop, Unreg1),$
  $(\text{"}payment\text{"}.equals(itfName), nop, Unreg2)]$,
  $\delta(Unreg1, GetService) = [(\text{"}reservation\text{"}.equals(itfName), nop, linked)]$,
  $\delta(Unreg2, GetService) = [(\text{"}payment\text{"}.equals(itfName), nop, selected)]$,
  $\delta(paid, UnRegister) = [(\text{"}reservation\text{"}.equals(itfName), nop, Unreg3)]$,

```
GLOBAL{
    VARIABLES{ ... } EVENTS{... }
    PROPERTY reservation {
       %% Class property for interface "reservation"
        STATES{...} TRANSITIONS{ }}
    PROPERTY payment {
        %% Class property for interface "payment"
        STATES{...} TRANSITIONS{ ... } }
    %% Introduction of this new keyword
  FOREACHCLIENT(Long pid, String itfName){
     %% Instance property.
    VARIABLES{ ... }
    EVENTS{%%framework and method invocations events:
       %%method invocations event definition
       selectFly={reservation.selectFly(String)}
       ...}
    PROPERTY clients {
       STATES{ STARTING{ linked{}; }
               NORMAL{selected{} ... }
               BAD{...} }
       TRANSITIONS{
               linked -> selected [selectFly//]
               ...   }
  } } }
```

Figure 3.11: An OSGiLarva property description file with the **global** keyword associated to two interfaces properties and FOREACHCLIENT keyword

$$\delta(selected, GetService) = [(\text{"}reservation\text{"}.equals(itfName), nop, linked)],$$

$$\}$$

In this OSGiLarva automaton example, a "true" is meaning that this transition has no condition, the "nop" means no any action in this transition. A graphical view of this property automaton is shown in Fig. 3.12.

In order to show a complete use of the OSGiLarva system, we propose to observe two execution traces t_0 and t_1. They are chosen such as one is correct and the second one is error ending. Since the value of the context variable "itfName" is provided by LogOs automatically, we add it as a subscript of input events when needed.

- t_0=[$reservation.selectFly(String), payment.pay(String),$
  $reservation.confirm(Boolean)$]

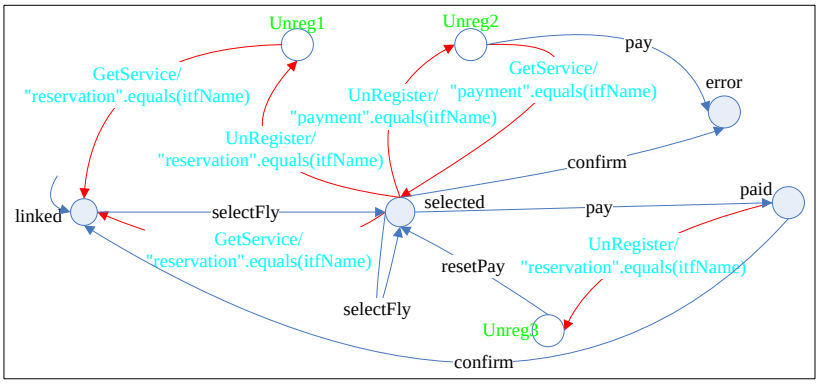


Figure 3.12: An OSGiLarva clients-side automaton of the airline reservation

- t_1=[$reservation.selectFly(String), UnRegister_{itfName="payment"}$, $payment.pay(String)$]

In order to use OSGiLarva to check these traces, we first have to translate these traces as event traces according to the events clause of the property (i.e., Fig. 3.13), giving the following events traces t'_0 and t'_1, playable on the automaton property, as defined in **Definition 4**.

Let t'_0 and t'_1:

- t'_0=[$selectFly, pay, confirm$]
- t'_1=[$selectFly, UnRegister_{itfName="payment"}, pay$]

by playing them on the property A, we then obtain the following logs:

- $check(t'_0, A)$=[]
- $check(t'_1, A)$=[$IdOfClientMakingThisError$]

Since no bad state occurs while playing t'_0 on this OSGiLarva property automaton, t_0 is correct and the obtained log is empty. Conversely, the bad state $error$ is visited by t'_1 with the transition $\delta(Unreg2, pay) = [(true, nop, error)]$. So t_1 is an error ending execution and the identification of the client making the error ending trace is added in the monitor's log.

Finally, we give a third execution trace t_2 which is also checked by OSGiLarva with a translated events trace t'_2. There is no bad state reached in t'_2. So t_2 is correct. But in fact, a problem is hidden in this example: after reservation service unregistering (i.e., S_1), a new service is also gotten (i.e., S_2). OSGiLarva system can't check whether the following method $selectFly$ is invoked through a stale reference of the unregistered service (i.e., S_1) or not. A solution of this problem will be proposed in Chapter 5.

```
EVENTS{
    selectFly = {reservation.selectFly(String)}
    confirm = {reservation.confirm(Boolean)}
    pay = {payment.pay(String)}
    resetPay = {payment.resetPay()}
    UnRegister = {UnRegister}
    GetService = {GetService}
}
```

Figure 3.13: EVENTS description in an OSGiLarva property

- t_2=[$reservation.selectFly(String)$, $payment.pay(String)$, $UnRegister_{itfName="reservation"}$, $payment.resetPay()$, $GetService_{itfName="reservation"}$, $reservation.selectFly(String)$, $payment.pay(String)$, $reservation.confirm(Boolean)$]

t'_2=[$selectFly, pay, UnRegister_{itfName="reservation"}$, $GetService_{itfName="reservation"}, selectFly, pay, confirm$]

3.4.2 Implementation

Since our monitor implementation is based on both LogOs and Larva systems, we first present more useful details about both systems. We then present our adaptation of LogOs to intercept service interactions and give some details about our modifications of Larva.

3.4.2.1 LogOs system

In this section, we introduce more details about LogOs system [47]. This will be helpful to describe our modification on it in our OSGiLarva implementation (i.e., section 3.4.2).

LogOs Structure In LogOs system, there are four parts be composited: Annotations, Interception, Logging, Storage. Each part has its responsibilities. Annotations part aims to define Domain-specific language (DSL) through Four Java Annotations that modify service interface method call logging standard behaviour. Interception part is in charge of calls interception during system execution. Adding the defined annotations in the service interface. When the specified service method or method parameters are marked in the interface side, all these will be intercepted during running.

Logging part is for tracing the intercepted actions or parameters and recording these information details. It includes: unique id, time, method name and so on. Storage part is responsible for output the traced record for users. The record file is named like logosng-recorded-on-1368691248161.cur. It means that this file records all intercepted service method invocations from the framework time stamp 1368691248161 until system stopped.

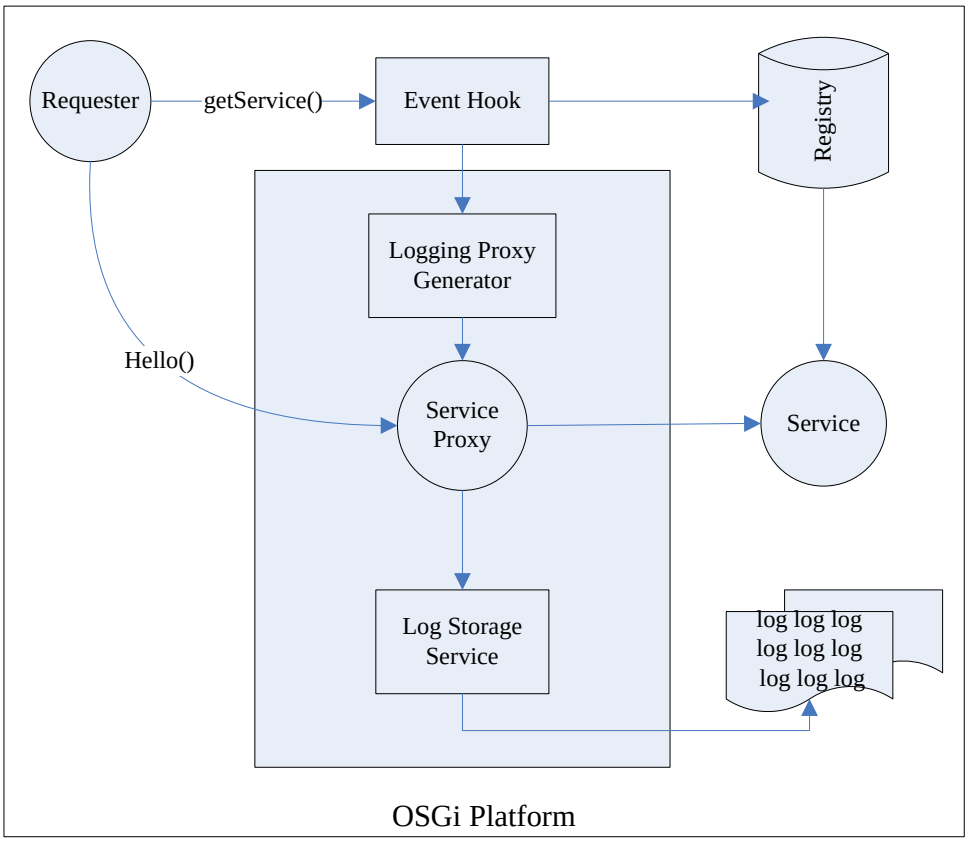


Figure 3.14: Processing of LogOs system works for system based on OSGi framework

Implementation of LogOs architecture on OSGi LogOs system is working in a system based on OSGi framework (in Fig. 3.14). LogOs system uses EventHook package for building interception proxy. Once a service proxy built for one registered service, the other services must invoke its service methods through this built service proxy. This invocation is indirect. This service proxy can capture log events for storing.

LogOs system can observe all invoked service methods which are defined in its

service interface with java annotations, even if a being used service is unregistered and a new service appeared.

3.4.2.2 Larva Tool

In this section, we also introduce more details about Larva tool for ease understanding our modification on it in our OSGiLarva implementation (Section 3.4.2).

Larva Syntax For this Larva tool, property description for the monitored system is a key point. This property description language aims to describe a Finite-state-machine(FSM) to verify the running software system. There are four parts need to be declared in a property description file: EVENTS, VARIABLES, STATES, TRANSACTIONS. We introduce the syntax of each declaration part at here. For ease understanding the principle of property description in Larva, we use that example which is explained in section 3.2 to explain the four declaration parts.

1. EVENTS: Firstly, we need to explain the monitored events in the property description. All the defined events in a property file is in the light of thesse methods from this example, for example, Auth(), Lock(), SomeUse(), UnLock(), and UnAuth(). They are defined like this: `eventName()=*.methodName()`. The `"eventName"` is an identifier arbitrary chose by designers and used in transitions. The `"*.methodName()"` is a real method of the monitored software system and defined for AspectJ technology, to insert a calls of interception in the monitored system. The equivalent event declaration part in a property file is showing in the code snippet Fig. 3.15.

```
EVENTS{
%% Property designer needs to
%% express how to retrieve the
%% identifier:
   Auth() = {*.Auth()}
   Lock() = {*.Lock()}
   SomeUse() = {*.SomeUse()}
   UnLock() = {*.UnLock()}
   UnAuth() = {*.UnAuth()}
}
```

Figure 3.15: EVENTS description in a Larva property file

2. VARIABLES:

The FSM is used in Larva property description includes variables. They are updated by actions and are used in conditions in transitions. Variable is defined like Fig. 3.16.

```
VARIABLES{
   int Cnt = 0;
 }
```

Figure 3.16: VARIABLES description in a Larva property file

3. STATES:

   State is a necessary part in FSM. In Larva property description, the responsibility scope of state is fixed. They are: Accepting, Bad, Normal and Starting. We can define our states under these fixed states. Under Accepting, our states be taken as the system desirable state to terminate. Conversely, states under Bad set correspond to an error. Under Normal set of states, all states are middle state during system running. Finally, only one state is starting. It defines the system starting state. An example of the states syntax code is showing Fig. 3.17.

```
PROPERTY cients{
   STATES{
        BAD{bad}
        NORMAL{ok}
        STARTING{start}
    }
}
```

Figure 3.17: STATES description in a Larva property file

4. TRANSITIONS:

   For express transitions, we need to use the described EVENTS, VARIABLES, STATES. The syntax of a transition is: $startState \rightarrow reachedState[event \backslash condition \backslash action]$. From one state to an other state in terms of $event$ and $condition$, execute the $action$ before arriving $reachedState$. $condition$ and $action$ are optional elements, the rest of elements are necessary for expressing a transition. We design some transitions according this syntax in the following Fig. 3.18.

```
PROPERTY clients{
   TRANSITIONS{
      ok -> ok [Lock\Cnt=0\Cnt++;]
      ok -> bad [Lock\Cnt=1]
      ok -> ok [UnLock\Cnt=1\Cnt=0;]
      ...
    }
}
```

Figure 3.18: TRANSITIONS description in a Larva property file

Existing Larva Property Description Language A Larva property description file can contain several automatons. The file is structured in terms of contexts. The global context can contain several properties and each of them can introduce a new context. A context is defined by variables and listened events. Each inner context can access the global variables.

A FOREACH structure allows a property to be instantiated for each different value of an element, considered as an identifier. Channels can be used by automatons to communicate together. These channel-generated events are broadcasted to the current context and below. So, if two inner contexts need to communicate, they can do it through channels. Clock Larva property description language allows real-time properties, by the use of clocks. Clock can track an event is timeout or not after a fixed timed. So it is a tool for keeping some real-time application security through timed.

A generic structure of a Larva property file is given in Fig. 3.19. It shows a file containing two properties: a global one and an instantiated one.

Both tools (LogOs system and Larva system) are useful for its corresponded environment. But for each other environment, there exist some disadvantages. The LogOs system can capture specified actions with dynamicity from a running system. But it can't check whether the captured action is authorized or not. The Larva can check specified properties. However, it does not have any dynamicity resilience characteristic. In following two sections, we will give our adaptation of LogOs system and Larva tool to resolve these problems.

3.4.2.3 Adapted both LogOs and Larva systems

LogOs is a transparent logging toolkit for the service activity inside the OSGi architecture. As soon as the LogOs bundle is started, each service registration is observed by the system. Thanks to the OSGi hooking mechanism, a LogOs proxy can be generated between the service and its consumer. Hence, every method call, including

```
GLOBAL{
   VARIABLES{ ... }
  EVENTS{ ... }
 PROPERTY P1 {
    STATES{...}
   TRANSITIONS{ ... }
  }
 FOREACH (Object u ){
      VARIABLES{ ... }
     EVENTS{
          %% Property designer
             needs to
          %% express how to
             retrieve the
          %% identifier:
          someEvent() = {*.method}
          ...
      }
      PROPERTY P2 {
         STATES{...}
         TRANSITIONS{ ... }
      }
 }
}
```

Figure 3.19: Generic larva property file with two properties of two types

parameters and returned values, are automatically intercepted.

For each event captured by a LogOs proxy, a corresponding LogOs event-description is forged and propagated to LogOs. In our adaptation, LogOs proxy forwards them to the associated monitors.

We have extended LogOs annotations to enable the user to declare whether an interface is to be monitored or not. If an annotation is present, the monitoring class is loaded when a service implementation is registered.

Moreover, LogOs integrates a mechanism to observe services registration, which is originally used to generate service proxy at load-time. This information is sent to the Larva monitor.

In section 3.4.2.2 and section 2.2.3.3, we introduced the syntax and structure of Larva tool. The advantage of this tool is a monitoring of real-time property. The weakness of this tool that needs to use apsectj technology to weave interception calls in the monitored system at coding-time or compiling-time or loading-time. In dynamic SOA

systems, it doesn't work when service dynamic substituted during runtime. Because the aspectsj technology is used in Larva system.

We adapted Larva to OSGiLarva by removing the part associated with the injection of aspects. In order to replace this part by a call from LogOs, we need to make this provided call by LogOs be checked in the generated Java code which describe multiple service interfaces' properties and clients' properties. For checking a call among multiple service interfaces, we also specify a format of a provided call description in LogOs associated to the events definitions in property description, like: $interfaceName.methodName(type, ...)$. $type$ list are method's parameters types. In order to consider dynamic events in described properties, we introduced some new primitives (Section 3.4.1.1: Considering dynamic primitives) in the property description language which are generated by the latest version of LogOs correspond to events definitions.

3.4.3 Registration of a service providing specification

We propose to enable the declaration of properties to monitor as a part of OSGi bundles, as shown in Fig. 3.20. Indeed, an OSGi bundle is an archive providing four elements: a collection of *interfaces*, a collection of *services implementations*, *bootstrap code*, and *POM.xml*, which is called when loading or unloading the bundle. Thanks to the OSGi architecture, service interfaces, service implementations, bundles and the manifest file POM may have different life cycles depending on the deployment scheme, since interfaces may be deployed with a bundle other than the one containing the service implementation.

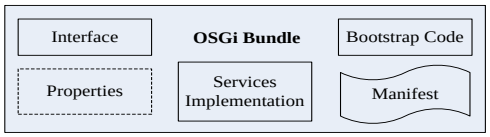


Figure 3.20: Structure of an OSGi bundle providing properties

As such, we propose to keep the same philosophy when providing properties. We consider that they can be either provided by the same bundle as implementation or by another one. Since interfaces are typing specifications of services and OSGiLarva Class-Properties are behavioural specification of services, it makes sense to map the life cycle of class properties to that of interfaces. On the other hand, the Instance-Properties life cycle describes the behaviour of all service interaction and thus it makes sense to map its life cycle to the client-service connection life cycle.

3.5 Evaluation

In this section, we present some benches of OSGiLarva. There are mainly two implementations used for executing OSGi services: Apache Felix and Eclipse Equinox. In our benches, we use the current Apache Felix which is an open source implementation of the OSGi Release 4 core framework specification, on the top of the Java 1.6.0-06 Virtual Machine. The machine used for these tests runs on an Intel Pentium M at 1.4GHz CPU with 640MB of RAM and running under Gentoo 4.2.3 with 2.6.22-gentoo-r8 kernel version.

In the following, we are using two examples: one without dynamicity and another with dynamicity. Indeed, since we will make efficiency comparisons against Larva, which does not support dynamicity, we then need to have a static example. This example is just a loop making some calls to a function provided by a service. On the other hand, the dynamic example is very close to the one described in Section 3.2, but with a loop on the client side. This loop specifies the concrete actions from the client and contains a call to a service, followed by an unregistration of the service, a get service to have a second service, a second call, and finally a new registration of the unregistered service. In our benches, we modify the amount of loop iterations to study the variation of the time cost in the long run and its variation due to JIT compilation.

We made three kinds of tests to study performances of OSGiLarva: a comparison between the execution time of OSGiLarva and Larva, a comparison between the execution time of OSGiLarva and OSGi, and a comparison between the execution time of OSGiLarva and a Instance-Property-only in OSGiLarva. Indeed, we hypothesized that the identification of the client (and hence the Instance-Property) is a bottleneck, but benches show that it is not so costly.

Here is the definition of some keywords appearing in this section:

- Larva: the time cost from the example with the original Larva system.
- OSGiLarva: the time cost from the example with the OSGiLarva tool.
- WithoutOSGiLarva: the time cost from the example running under OSGi, but without any monitoring system.
- OSGiLarvawithoutPID: the time cost from the example with a weaker version of OSGiLarva where we removed the generation of a caller Id from the system.

Finally, for each test, we made two curve charts. The "Time cost comparison" curve chart shows amount of loop iterations on the horizontal axis, and time cost in milliseconds on the vertical axis. The "Cost ratio" curve chart shows amount of loop iterations

on the horizontal axis, and change ratio of time cost in percentage points on the vertical axis. The cost ratio is calculated by the time cost of the example with the monitor divided by the time cost of the example without the monitor.

3.5.1 Monitoring cost by using a proxy (OSGiLarva VS Larva)

The goal of this test is to evaluate the performance of OSGiLarva (with a proxy) and to compare with the one of the Larva tool (with AspectJ) on the same functions example. Since Larva does not support OSGi dynamicity, we made the comparison on a example without loading of services. In this kind of comparison, we just use the two tools to monitor the normal events from the communication of client using services.

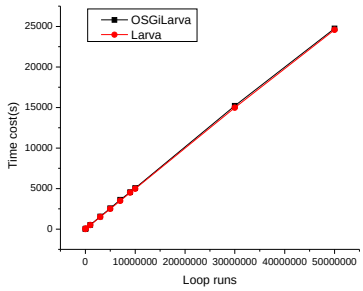


Figure 3.21: Comparing time cost of a static example with OSGiLarva and Larva

Fig. 3.21 is a comparison of the time cost in the execution of a static example with Larva and OSGiLarva monitors. We can observe that both curves are very close. Hence, OSGiLarva does not add too much cost by its proxy approach.

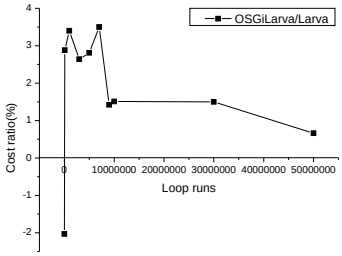


Figure 3.22: Comparing cost ratio of a static example with OSGiLarva and Larva

In order to be more precise, in Fig. 3.22 we plot the curves of the cost ratio between Larva and OSGiLarva time cost results. The change ratio of time cost is lower than 1%. This change ratio is from the proxy in OSGiLarva. Thanks to this proxy, OSGiLarva can make the behavioural monitoring bindings dynamic and loosely coupled. The pre-condition of this test is that the monitored service is never replaced by another one. If the monitored service is replaced during runtime, Larva will not be able to detect any of its events. But OSGiLarva can continue to monitor it.

Since these two technologies are not using the same Virtual Machine, the JIT is also not the same. We think that this difference is the explanation for the behaviour observed in the first run, which is stable and always faster on OSGiLarva. This difference is probably also the explanation for diminution of the overhead when the loop is longer.

3.5.2 OSGiLarva efficiency (OSGi VS OSGiLarva)

This test runs the dynamic example described as a running example in this article, but with a loop inside the client. We then run it with and without OSGiLarva in an OSGi environment. It aims to evaluate the raw impact of OSGiLarva on service invocation and service events from the framework. The property events includes normal events and framework events.

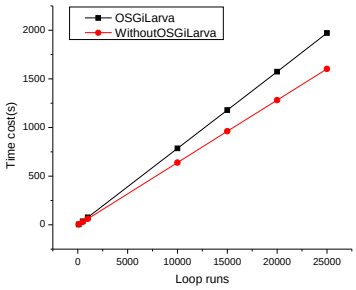


Figure 3.23: Comparing time cost of the case study example with and without OSGiLarva (simple method in service side)

From Fig. 3.23, we know that the performance impact of OSGiLarva is stable at around 23% on this example.

For every monitored service invocation and framework events, OSGiLarva performs its indirection work: it verifies the actions from the original system and computes the current client id, and finally it outputs the monitored traces to the developer or the user at real-time. The cost ratio almost becomes a horizontal line shown in

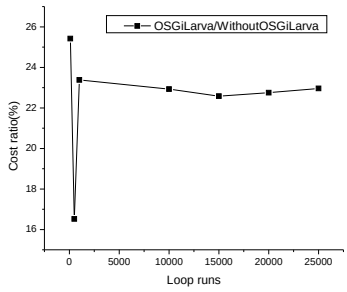


Figure 3.24: Comparing cost ratio of the case study example with and without OSGiLarva (simple method in service side)

Fig. 3.24, except for the two first points at about loop 100 runs and 500 runs. We presume that it is the initialization of the JIT which is causing this anomaly.

It is important to note that this 23% overhead is a metric including the call of methods events and the framework events. The biggest part of this overhead is associated to the cost of generating a new proxy and placing it in front of newly requested service.

3.5.3 Overhead associated to getting the caller id

In order to associate each communication to the right client in Instance-Properties, we compute a caller Id. However, we get it through the $SecurityManager$ which is a non-internal way of finding the caller class and caller Id. As such, one would expect extra time costs because of the $SecurityManager$, warranting further investigation.

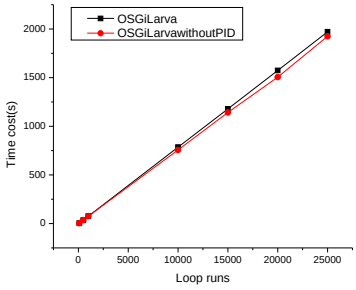


Figure 3.25: Comparing time cost of the case study example with OSGiLarva but with or without client Id

Thus, the following test is just for knowing the performance impact from compute current caller Id during runtime. We then compare the cost of the Case Study with

and without the Instance-Property and then, with or without getting the caller Id.

From Figs. 3.25 and 3.26, we observe that the time cost of the two kind of monitoring are very closed. The impact cost is lower than 5%.

Indeed, in such a simple test example, the body of the called methods are very small. Hence, the most of the time cost is from invocation itself. So, if the service method is a more complex and real one, the time cost for getting caller id and caller name will far less than 5%.

Moreover, even at 5% time cost, which actually boils down to 1.15% (5% of 23%) of the system runtime, we conjecture that it is an acceptable price to pay for obtaining the crucial information for identifying which client is currently using a particular service.

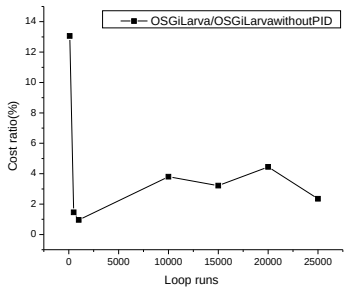


Figure 3.26: Comparing cost ratio of the case study example with OSGiLarva but with or without Client Id

3.6 Summary

In the highly dynamic environment of the SOA, where software can be replaced on the fly at runtime, the challenges for ensuring correct behaviour increase as the software has to be checked at runtime. In this context, we have identified two properties, that we consider are required to make a dynamic monitor for dynamic SOA systems: (i) *resilience to dynamicity*, i.e., the monitor is able to maintain state even if the service implementation is substituted at runtime, and (ii) *comprehensiveness*, i.e., that no implementation of the service interface can bypass the monitor's observation.

We have instantiated the approach in the context of the OSGi framework through a preliminary implementation, OSGiLarva, which integrates an adaptation of two existing tools: Larva and LogOs. This OSGiLarva monitoring system is freely downloadable at: https://github.com/Yufang-DAN/OSGiLarva-monitoring-system. Similar to Larva, OSGiLarva accepts the Larva property description language as input, hence

inheriting all its features, including its expressiveness and its readability for non-expert users. Furthermore, it enables the description of both class properties and instance properties. This feature has been instrumental for OSGiLarva to monitor both properties which span the whole duration of the interface life cycle, and the individual client's point of view of the service, possibly spanning over different implementations of the service requests. We have also extended the Larva event description language, in order to consider not only calls or return of method calls, but also OSGi framework events such as the registration of a service or its request by a client; this has been achieved by introducing reserved event names which are usable transparently as if using standard method calls. Moreover, we gave the rules about describing one property file or multiple property files to possibly several interfaces. And then this OSGiLarva system can monitor clients use one or more services from multiple interfaces.

As observed in section 3.5 about monitor the communications between clients use services from one service interface, our approach is not so inefficient when compared to injection-based monitoring tools like Larva. While our approach is based on an OSGi hook observing all occurring events instead of aspect-oriented programming, the extra cost is small: tending to less than 1% increase in overheads. Since, this approach is crucial for dynamicity resilience, the cost incurred seems to be a reasonable.

An interesting element of this approach is its non-intrusive aspect. Indeed, in contrast to the aspect-oriented approach, we keep the original byte-code unchanged. This property can be useful if we want to switch off a monitor or be able to check the binary signature of the code as an authentication credential [41].

The notion of comprehensiveness also has a number of benefits since anybody with some privileged access to the platform (user, developer, or service) can define a behavioural property and ask the system to check if services respect it. This can be done for many reasons, such as: debugging deployment, privacy concerns, or to learn about typical usage patterns of a service.

Finally, in order to make the OSGiLarva more autonomous and keep services' atomicity, we enable the framework to associate one property file to possibly several interfaces. All properties descriptions of these service interfaces are independent described in **global** clause. If there exists an order of clients' methods invocations among these services interfaces, we can express this property in FOREACHCLIENT context to check it.

4

A Safe Service Use Layer to Deal with Dynamic Service Disappearance

The service oriented approach is a paradigm allowing the introduction of dynamicity in developments. If there are many advantages with this approach, there are also some new problems associated to service disappearance. The particular case of service substitution is often studied and many propositions exist. However, proposed solutions are mainly server-side and often in the context of web-services. In this chapter, we propose a client side safe service use approach to allow service substitution without any restart of the client and without any assumption on external services. Our proposition published in [90], is based on a transactional approach, defined to automatically and dynamically substitute services, by preserving the current run and collected data proposed.

4.1 Introduction

The dynamic service oriented approach is a paradigm introducing loose-coupling into software architectures. A developer can simply choose an API describing requested

service and develop its software without knowing which implementation will eventually be installed on the final client system. Currently, most popular uses of this approach are done by web services, Enterprise Java Beans (EJBs), Android systems and the OSGi framework. Main studies are about Web services, but with the server-side point of view [46]. It means that the service provider can make many assumptions on provided services with the objective that a service substitution can be done without any consequence on the client, even if the service is state-full. State-full services are the one that maintain internal state across successive invocations from the same requester.

We propose to study the client point of view, and we base our proposition on the OSGi framework [9]. We are focusing on the case of a mobile platform with OSGi that can discover or lose connection to some service providers. In such a case, a service requested by a client can be lost while in use. Hence, we cannot make any hard hypothesis on services lifetime, but we can propose some good practices in client development in order to be resistant to the substitution of services. The service substitution is well known as a self-healing software technique [51, 37]. In this thesis we will propose a solution to make services more self-heal through client side without modifying client's code when service is unloaded.

If a service is unloaded, then the main problems are:

1. to detect the service unloading;
2. to choose a new service ;
3. to load the new service by preserving the internal state of the unloaded one.

We don't want to focus on the service selection problem, since this problem has been largely studied elsewhere in some literatures which have been explained in Section 2.3.2. We will focus on the two other problems.

In this chapter, we propose a *safe service usage* (SSU) approach for the development of the client, inspired from the transactional approach of concurrent systems. We will consider the problem of detecting unloaded service and one of the new services loading. In our proposition we will first consider the easy case of using a single service, before introducing the substitution of a service while using a set of state-full services.

If a software is developed by using correctly this SSU approach, we guarantee that it can, according to its preferences:

1. be actively notified of the unload by a specific exception,
2. continue its execution with a new service that has been automatically substituted, even if the service is a state-full one, with a very light overhead of code to write.

We also claim that the development cost is low in comparison with the development cost of a similar software with the same capabilities but developed without this SSU approach. Finally, we will show that our approach does not restrict the expressiveness of developed software, which means that every program using service can be rewritten to use the proposed SSU approach.

This Chapter is organized as follows: Section 4.2 introduces the context of this work (OSGi) needed to explain the problem by an example. Section 4.3 describes the contribution of this Chapter, about service substitution. It also gives a discussion about the expressiveness restriction and the good use of the provided SSU approach. In order to fix the global understanding of the reader, Section 4.4 describes the tool developed to show the feasibility of the approach. Finally, Section 4.5 concludes this work of this Chapter.

4.2 Example

In Section 3.2, we presented an example model of a dynamic system which is monitored with respect to our proposition. At here, we will use Airline Reservation system (i.e., Fig. 3.10 in Section 3.4.1.4) to express the ideas proposed in this Chapter.

The correct use of this Airline Reservation system is like that: step 1 select its fly, step 2 pay the reservation and final step is to confirm this airline reservation. This example (Fig. 4.1) illustrate the fact that: if a single client will use multiple services at same time, before invoking $confirm(...)$, the service reference of S_1 will become stale if S_1 is unregistered. For avoiding this issue, we propose that an exception be thrown or that a new service replace it when stale reference occurs.

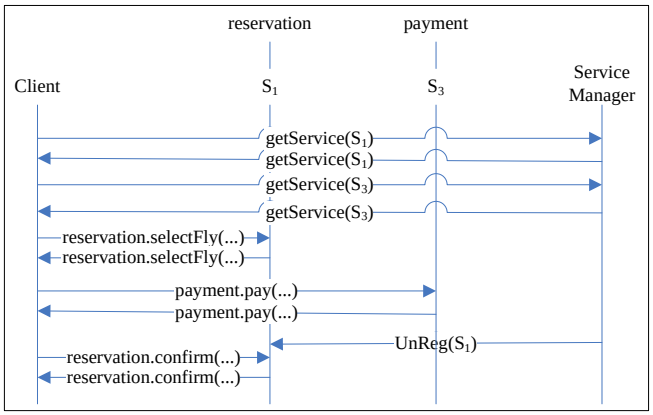


Figure 4.1: Stale reference occurs in Dynamic SOA system

These propositions are detailed through two execution scenarios of this example in Fig. 4.2 and Fig. 4.3.

In the scenario from Fig. 4.2, S_1 unregistered between two calls of $pay(...)$ and $confirm(...)$ and there does not exist any service to replace it. During $confirm$ is invoked, the proposed SSU layer will roll back the $resetPay()$ and then throw an exception to inform the invoked client that the service reference of S_1 is stale.

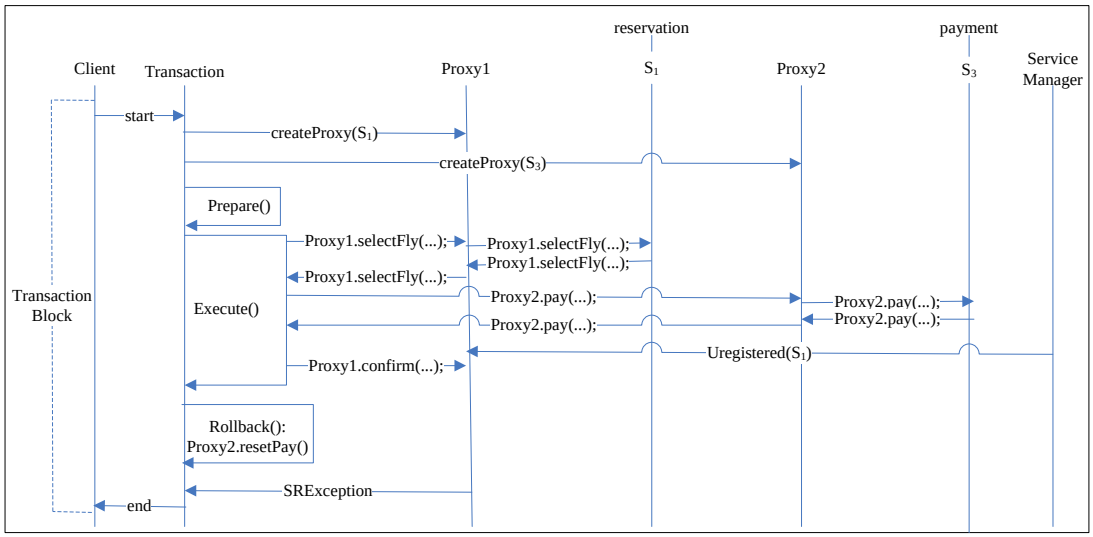


Figure 4.2: Example of scenario with Exception to handle stale reference

In order to illustrate the second approach, based on service substitution, we assume that S_1 and S_2 implement $reservation$ and S_3 implements $payment$ in the scenario from Fig. 4.3. We then express that: at the beginning of the transaction of this client airline reservation, we first need to create proxies for the two service interfaces. These proxies are used to check whether service is unregistered or not and take corresponding measures to prevent the use of stale reference.

During the transaction execution, if S_1 is unregistered before invoking $confirm(...)$ and then a requestion of $confirm(...)$ from client is sent to $Proxy1$, the $Proxy1$ can check if the current used service (i.e., S_1) is unregistered. This transaction block executes $Rollback()$ to revert the used related services (i.e., execute the method $resetPay()$ of S_3 through $Proxy2$ in this transaction method $Rollback()$), and then the $Proxy1$ makes a substitution (i.e., S_2 substitutes the unregistered S_1). After service substitution, this transaction block is re-executed from its transaction method $Prepare()$ until the end of the $Finish()$. Finally, this transaction block returns the results from the transaction method $Execute()$ to the client.

This example explains that an OSGi service working with our proposition can

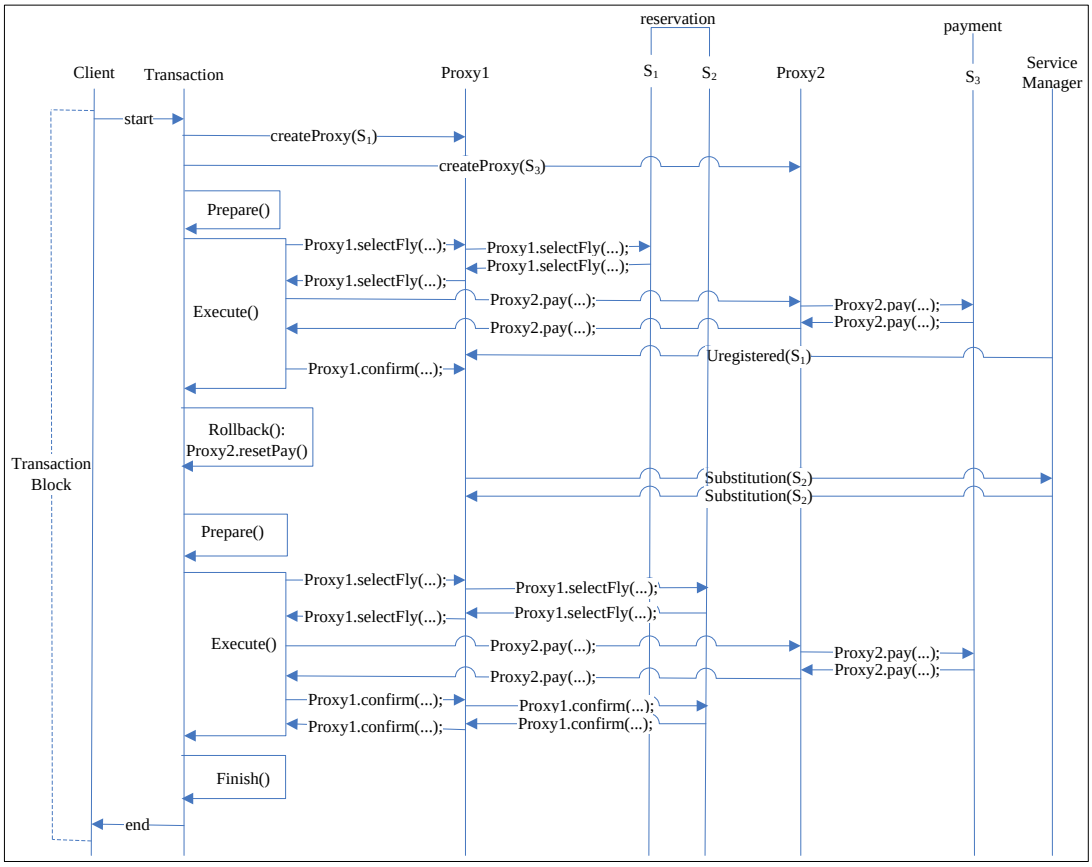


Figure 4.3: Example of scenario with service substitution

avoid stale references. The transaction approach can make coherence of multiple services cross usage in one processing.

4.3 Contributions

We propose to add a SSU layer into the OSGi framework, in order to make systems being more fault tolerant. This SSU layer has been used by clients to make it unload service aware. To describe what have to do this SSU layer, we firstly introduce usual approaches of fault tolerant systems. Next, we describe our solutions for the simple case where the client use a single service. Finally, we extend solutions to take account the case where several state-full services are in use at same time.

4.3.1 Fault tolerant technology as a fundation

Our proposition aims at making OSGi-based system automatically handle services dynamicity in order to avoid null pointer reference and stale references effects at runtime. In this section, we will explain how to use fault tolerant technology in OSGi-based system for handling occurred faults. Usually, fault tolerant systems whose execution can continue to deliver correct service even if a fault occurs. In such a system, the first problem consists in identifying that a fault occurs [11]. In our proposition, we define precisely what is a fault: the unload of a used service. And we will detect it by a listener which is added at the time of service registry.

As introduced in background section 2.3.1, there is usually three families of treatment to recover an error [37]:

1. to mask the error;
2. to roll-forward in the execution until a new stable state is reached;
3. to roll-back to the previous stable state and restart the execution from it.

Usually, the mask an error mechanism depend on redundant information provided by the system. Since we can not have it, we will focus on the two other treatments. We propose a model associated to the last two treatment families. In order to implement the roll-forward mechanism when a service disappears, we propose to throw an exception that explicitly advice the client that the service is no more available. This mechanism is like the scenario described in Fig. 4.2. A *try and catch* mechanism could then allow to reach a new stable state in forward. Finally, to implement the roll-back mechanism when a service disappears, we propose an automatic substitution of the service by another one which implements the same service interface. This service substitution will be state-full service resistant as described in scenario of Fig. 4.3. Indeed, the service substitution doesn't need any restart of the client and any assumption on external services through using a transaction mechanism.

In the following, we present these solutions in the context of a single service use or a multiple services use.

4.3.2 Safe OSGi Service Reference - Single service

When a service is unloaded, its instance is kept in memory until the garbage collector dispose it, then while there is at least one reference to it. However, both the Java language and the Java virtual machine specifications do not support a notion of *"volatile / dynamic"* references [52]. References to object instances cannot be changed "under the hood" unless explicitly re-assigned as part of a program control flow. This means that

encoding a thread-safe and dynamic-aware behavior of service references need to be captured as part of a proxy indirection.

4.3.2.1 Proxy Indirection

A very common pattern for transparently mediating interactions between client code and a component in object-oriented languages is the introduction of a *proxy object*. They are most often used to enrich existing classes with cross-cutting concerns code such as logging, security or remote object exposition. A good example are the *Enterprise Java Beans*, where developers write simple Java classes, and EJB containers enrich them with support for security, transactions [93] and other useful features. In our context, we will try to add some enrichment at client-side for service substitution that is transparently for client.

4.3.2.2 Proxy Requirements and Functionalities

The requirements for an OSGi service proxy depends on the usages. Hence, two kinds of policies and then requirements can be defined: Roll-forward policy and Roll-back policy. In a *Roll-forward policy*, method invocations must throw an unchecked exception if the underlying service reference is staled. The client itself just need to take account the possibility of such exception.

In a *Roll-back policy*, when a method invocation reached a stale reference problem, we will try to transparently replace the unloaded service by another service, and then to make the invocation on the new service. However, if the unloaded service is statefull, the substitution can be the source of many unexpected problems. We then need to replay a part of the last commands. For instance, if the service need to be logged in, when the service is substituted, the login method has to be invoked again before any other use of the service. The part of the code that the SSU layer need to re-execute is called a *transaction*. Transactional systems have been widely studied for several classes of problems and applications [5]. The type of problem that we are tackling is actually close to a transactional memory [56]. However, the service and the client are developed by knowing that if a transaction fails, then it can be executed again. Our proposition is an adaptation of these existing results in the context of OSGi, where services are developed without knowing that such a substitution can occur. The client is the only one knowing this.

Since the SSU layer need to know precisely which part of the code has to be executed again in case of substitution, then the designer of the client must declare a part of code as the transaction. However, this code can be executed many times, since many substitutions can occur. Hence, this code has to be pure. It means that no side

effect has to be done in the client by the transaction. In order to make it transparent for the client designer, this transaction method will be directly called by the proxy, as a callback method.

Finally, here are the requirements we identified as sufficient in the case of using a single service:

- **Awaited Behavior:** When a method is invoked, if the underlying service reference is staled, then the awaited behaviors are the following, for each policy:
  - Roll-forward policy: unchecked exception is thrown.
  - Roll-back policy:
    - If no substitutable service: unchecked exception is thrown.
    - Else: substitution of the service, restarting the invocation from start of the transaction method.
- **Proxy Requirements:** It depends on policy:
  - Roll-forward policy: the client would consider the possibility of an exception for each service call.
  - Roll-back policy: the client must provide a pure method making the transaction.

4.3.3 Generalizing to the Invocation of Multiple services

While a pure transaction method is sufficient at the granularity level of a method invocation on a single OSGi service, generalizing the approach to the coherent execution of multiple state-full services is more involving. Indeed, consider a block of instructions where several services are being used, and having a strong requirement for that block to be executed with a stable set of non-stale OSGi references. Given that, we cannot make any assumption on concurrency and the possibility for service references to become stale in the middle of a block execution. We need to provide a more powerful transactional-like framework to execute such blocks which is transparently for client and without constraints on services' design.

4.3.3.1 Requirements and Assumptions

Coping with the traditional definition of a transaction, we assume that a *transacted block* is a portion of code invoking a set of services, and that the whole block shall be successfully executed as a coherent whole. However, by opposition with the case

of using a single service, we can generate side effects in used services. Hence, the transaction is pure only by the client point of view. In order to think about the transaction block working with multiple services, we need to define a service roll-back code for other services. Fig. 4.4 shows the transaction diagram for multiple services: if the execution step has error, the transaction block will roll-back to the before state of this execution for the other related services, and then make a service substitution for restarting this transaction block.

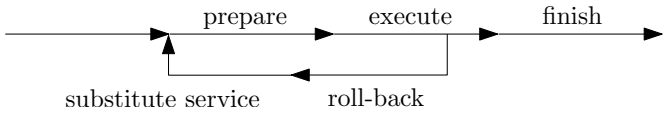


Figure 4.4: Transaction diagram for multiple services

Hence, in the context of multiple services, executing a transacted block requires:

- a declaration of the service interfaces it operates on,
- methods implementations to:
  1. put the transactional block into a coherent initial state before its execution,
  2. execute the actual block code,
  3. finalize work upon successful execution,
  4. compensate possible side-effects in other services, if a stale reference caused a failure during the execution of the block,
- a retrial policy to control how the block execution is attempted again when a stale reference caused a failure.

The context of an OSGi platform imposes very loose assumptions on the transacted block implementations. Especially, services in use are not aware of being used in a transactional context, unlike Java EE resources that implement transactional APIs. Consequently, the correctness of performing a compensation operation or the ability to retry a block execution greatly depends on such services suitability in such a context, and their public specifications.

4.3.3.2 Invocation Atomicity – a Correctness Hypothesis in a Multi-Processed System

The OSGi specification states that OSGi service event listeners is notified when a service is unregistered [9]. A service reference becomes staled when all event listeners

have been notified from the OSGi framework notification loop. Finally, we can take advantage of making a proxy to a service event listener in order to keep atomicity. Indeed, we can make a lock on the proxy object when performing a method invocation or when receiving a service un-registration event. This ensures a safe method execution as a reference cannot become staled in the middle of a method invocation.

4.3.3.3 Discussion

The generalization of the transacted execution of a set of services relies on strong assumptions:

1. services offer SSU proxies to compensate effects in case an aborted execution,
2. transacted execution blocks properly call compensation SSU proxies,
3. intended compensation SSU proxies are honored in service implementations,
4. services taking part in a transacted execution do not have further side-effects, or can compensate them if needed by the client.

In more traditional approaches, a transaction in SSU layer is designed for resources to be managed by a transaction monitor. In the case of OSGi services, this would be translated to service interfaces extending such an SSU layer, making it impossible to use other types of services even if they offered compensation capabilities. We instead opted for a more open approach even if incorrect transacted block implementations can easily defeat the intended purpose.

4.4 Implementation — A safe service use layer for OSGi

The contributions presented in the previous section apply not just to OSGi environments. Indeed, any service-oriented architecture is based on the assumption that a client code has no control over the services, including their availability and upgrades. We now detail how we implemented those contributions in OSGi in 2 steps. First we propose a simple service for building safe proxies to OSGi services, then we offer a service and an SSU layer for executing and defining transacted blocks. The interested reader can download the whole SSU layer and some examples at:
http://dynamid.citi-lab.fr/software/

4.4.1 Configurable Service Proxy References

4.4.1.1 Overview

Proxies can be created at runtime in Java by creating a class that implements the *java.lang.reflect.InvocationHandler* interface and passing it to *java.lang.reflect.Proxy* to obtain a proxy that is a subtype of one of more interface types. What we proposed here is a very simple and minimalist SSU layer API for generating proxies to OSGi services. It is exposed as an OSGi service of its own with the following interface:

```
public interface ServiceProxyBuilder<T>{
  public T getService(Class<T> c,
                      ServiceReference sr,
                      ProxyMode pm);
  public T getService(Class<T> c, ProxyMode pm);
  public T getFirstServiceMatching(Class<T> c,
                                   String filter,
                                   ProxyMode pm)
          throws InvalidSyntaxException;
  public ServiceBroker<T> getServices(Class<T> clazz,
                                      String filter)
          throws InvalidSyntaxException;
}
```

The interface mimics the OSGi service reference retrieval. *ProxyMode* parameters allow to specify whether a service reference becomes disabled after its using service has been unregistered, or if another available service can be used in place. This allows to cater for both stateless and state-full types of services:

```
public enum ProxyMode {
  DISABLED_AFTER_UNREGISTERED,
  RELOAD_AFTER_UNREGISTERED
}
```

A *ServiceBroker* is used when dealing with several services for the same interface.

```
public interface ServiceBroker<T> {
  public Set<T> currentServices()
          throws InvalidSyntaxException;
  public void discard();
}
```

It is really close to the OSGi service trackers, except that it has the following semantics:

- *currentServices()* returns a set of service proxies currently matching the service interface and filter specification,
- returned service proxies have the *DISABLED_AFTER_UNREGISTERED* proxy mode,
- *discard()* is equivalent to the *close()* method of an OSGi service tracker.

4.4.1.2 Usage

The following code, extracted from the tests suite that we defined along with our implementation, shows an idiomatic usage of the service proxy builder OSGi service, in order to be substitution resistant.

```
ServiceReference ref =
   bundleContext.getServiceReference(ServiceProxyBuilder.
      class.getName());
serviceProxyBuilder =
   (ServiceProxyBuilder) bundleContext.getService(ref);
EchoService service =
   serviceProxyBuilder.getService(EchoService.class,
      RELOAD_AFTER_UNREGISTERED);

for (int i=1 ; i<= 10000 ; i++) {
   assertThat(service.echo("plop"), is("plop"));
}
```

In order to make a short discussion about use and without use such SSU layer , it could be interesting to give an example without the use of the SSU layer in the following code block.

```
ServiceReference ref =
   bundleContext.getServiceReference(EchoService.class.
      getName());

EchoService service =
   (EchoService) bundleContext.getService(ref);

for (int i=1 ; i<= 10000 ; i++) {
```

```
    assertThat(service.echo("plop"), is("plop"));
}
```

The differences between the above two code blocks are that: for the former one, OSGi service can be dynamically executed substitution at runtime through this SSU layer without restart client's request when a being used service by client is unregistered. But for the later one, if system executed service substitution, client needs to restart and execute all its requests.

4.4.2 Transactional Block and Service Execution

4.4.2.1 Overview

We propose an interface which is an OSGi service to execute transacted blocks. This interface is as follows:

```
public interface TransactedServiceExecutor {
  public T executeInTransaction(
        TransactedExecution<T> execution,
        RetryPolicy retryPolicy)
      throws TransactedExecutionFailed;
}
```

To implement this block, we need to have a transacted execution to realize a specified transaction and need to have a retry policy to handle the rollback action from the executing transaction. Hence, the following two code blocks about TransactedExecution interface and RetryPolicy interface can realize them.

A transacted execution is specified through the following TransactedExecution interface:

```
public interface TransactedExecution<T> {
  public void prepare();
  public T execute();
  public void finish();
  public void rollback();
}
```

It uses a parametric type T which is the expected return value type of a transacted block successful execution.

Finally, the RetryPolicy interface is a simple interface which is notified of potential stale reference errors, and can in turn decide whether a further attempt can be performed. It can also be used to implement delays between retrials. An example would

be an exponential back-off delay over at most 10 executions. This interface is defined as follows:

```
public interface RetryPolicy {
    public void notifyOf(Throwable throwable);
    public boolean shouldContinue();
}
```

By the way, a possible "retry forever" policy can be implemented as follows:

```
public class RetryForeverPolicy implements RetryPolicy {

    @Override
    public void notifyOf(Throwable throwable) { }

    @Override
    public boolean shouldContinue() {return true;}
}
```

4.4.2.2 Usage

Given some fictious service interfaces *SomeService* and *OtherService*, an implementation of transactional block could be as follows:

```
private class SomeTransaction
    implements TransactedExecution<Void> {

  @ServiceInjection public SomeService someService;

  @ServiceInjection(type = OtherService.class,
                    proxyType = MULTIPLE)
  public Set<OtherService> otherReferences;

  @Override public void prepare() { }
  @Override public <Void> Void execute() {
    for (OtherService s : otherReferences) {
      s.doThis(someService.doThat());
    }
}
  @Override public void finish() {
    someService.release();
```

```
  }
  @Override public void rollback() {
    someService.undoThat();
  }
}
```

This above code block exhibits an execution of a transactional block for multiple services. It consists of 4 steps: prepare(), execute(), rollback() and finish(). Each step has its responsibility in this block that is described in Fig. 4.4. In execution(), it executes about the same method called by the other services. If this transational finish, this called service method is released. Or the transactional will cancel this execution for all the called services.

A more complete example would take greater care in the *prepare()*, *rollback()* and *finish()* steps. Annotations authorized in Java since Java 1.5. We can use java annotation to define some fields or methods what we need during development. Fields annotated with *@ServiceInjection* provided by this SSU layer API are injected with service proxies. The definition for this annotation is as follows:

```
@Retention(RUNTIME)
@Target(FIELD)
@Documented
public @interface ServiceInjection {
  Class<?> type() default ServiceInjection.class;
  String filter() default "";
  ProxyType proxyType() default SINGLE;
  ProxyMode proxyMode()
      default DISABLED_AFTER_UNREGISTERED;
  public static enum ProxyType {SINGLE, MULTIPLE}
}
```

It is used to configure how proxies shall be configured. Especially, they can have service reloading capabilities enabled, and they can support a single reference or a set of instances like it is the case for the *otherReferences* set in the previous example. An OSGi service filter can also be specified.

Finally, the block can be passed to the transacted executor service, which is also an OSGi service:

```
ServiceReference reference =
    bundleContext.getServiceReference(
        TransactedServiceExecutor.class.getName());
TransactedServiceExecutor transactedServiceExecutor =
```

```
        (TransactedServiceExecutor)
            bundleContext.getService(reference);
    transactedServiceExecutor.executeInTransaction(
          new SomeTransaction(), new RetryForeverPolicy()
           );
```

We used an optimistic approach. Having service proxies being injected into transacted blocks, we could have taken advantage of them to perform a giant lock spanning for the transaction execution lifespan. Indeed, it is possible to block the thread notifying that a service is going to disappear, thus keeping the reference valid until all receivers have been notified. Such an approach would avoid the need for rollbacks at the greater cost of limiting parallelism and breaking the OSGi framework requirements that service event notification handlers shall not block [9].

4.5 Summary

In this chapter, we proposed an approach and a tool that is freely downladable at http://dynamid.citi-lab.fr/software/ to make a service aware to the stale reference problem. If a software is developed by using correctly this SSU approach, we guarantee that it can, according to its preferences: (i) be actively noticed of the unload by a specific exception, or (ii) continue its execution with a new service automatically substituted, even if the service is a state-full one.

Main properties of this contribution are: (i) this solution is client side, (ii) it does not make any assumption on used services and (iii) it can be used even if used services are state-full services.

This contribution is based on the fact that the client designer knows how to use desired services. Hence, we do not try to compute which behaviors are authorized by a service. The client designer has just to make a normal use of the service and to propose a sequence to rollback in a stable state before to make another try with another service, in case of substitution.

However, if a rollback is done on the external service, a rollback would also be done on the client itself in order to hold in a consistent global state. Since such a development model is risky, we prefer to give the following guideline in the client development: *do not make any modification of the client state from its transactional part.*

As explained and illustrated in the OSGi core specification [91, Section 5.4: Stale References], to make a correct use of a service without stale reference can be a little bit tricky.

We then claim that the development cost of a service using our SSU approach is

low in comparison with the development cost of a similar software with the same capabilities but developed without our SSU approach. However, according to the simplicity of the proxy and the fact that OSGi programs do not have usually very frequent call of services, then we can consider as insignificant the time cost of such a proxy. The worst case is when many substitutions occur. But at this time, the time cost is the trade off for keeping a continuity in the use of the client. Indeed, as explained and illustrated in the OSGi core specification [**?**, Section 5.4: Stale References], to make a correct use of a service without stale reference can be a little bit tricky.

Moreover, we do not introduce any restriction to the expressiveness of services. In fact for the worst case, we can include the whole program in one transaction. Hence, if a service is substituted, then the whole program is restarted and fully executed with the new service. Hence, this SSU approach does not add any restriction in the software development.

5

A Dynamic Monitoring System with Fault Tolerance

5.1 Introduction

In the previous chapters, we firstly introduced *OSGiLarva*, a monitoring system for OSGi-based systems. Secondly, we introduced a *safe service use* (SSU) Layer, an API that aids the development of client-side applications coping with stale references.

In this chapter, we propose a new monitoring systems (NewMS) (Fig. 5.1) that merges OSGiLarva system and SSU Layer. It enables not only to support the systems' dynamicity on OSGi, but also to deal with stale references from SSU layer. A monitor sits between client and server and is inserted at client-server binding time. NewMS focuses on the following principles: *dynamicity resilience*, *comprehensiveness* and *fault tolerance*. Since the NewMS considers some new property events and property descriptions associated to the processing of stale references, it gains in monitoring precision while stale references occur.

To achieve this, there are some restrictions while directly using both systems together. The source of these restrictions is mainly coming from the use of two proxies chained in front of a single service: one proxy, named a LogOs proxy, is created for

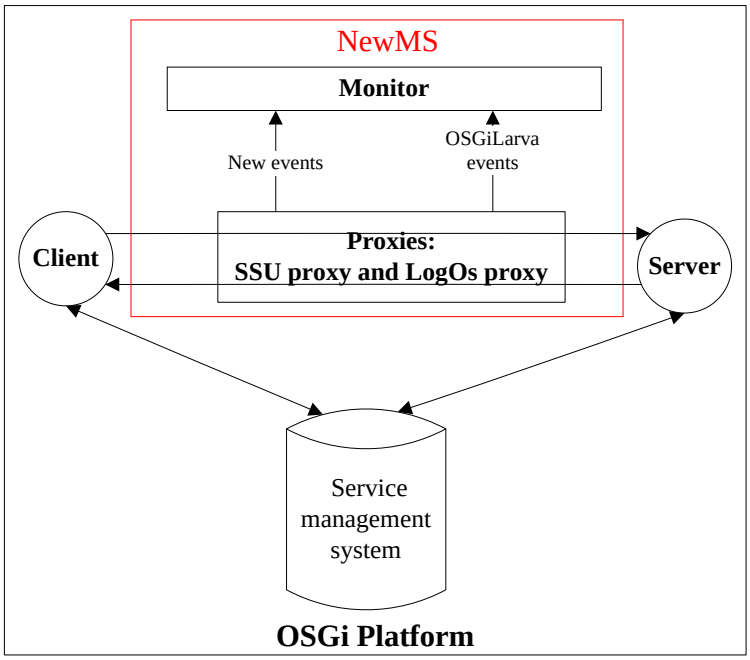


Figure 5.1: Generic architecture of the dynamic synthesized monitoring system

each registered service. Another proxy, named SSU proxy, is created at the client side in front of the service which is indeed the LogOs proxy. Clients communicate with the SSU proxy, the SSU proxy communicates with the LogOs proxy and the LogOs proxy communicates directly with the service. Hence, SSU proxy doesn't have any direct communication with the service.

If this service is unregistered, the SSU proxy might make method calls on a stale reference. Conversely, if a stale reference is detected by the SSU proxy that throws an exception, this exception can not be observed by LogOs proxy, since the exception occurred inside SSU layer system.

We describe our main propositions about the NewMS system in Section 5.2. Then, we give implementation guidelines for NewMS architecture in section 5.3 with resolutions of these restrictions, Finally, we express a summary of this Chapter.

5.2 NewMS generic expression

We first explain property events associated to stale reference integration in the NewMS system. Then, we show algorithms to translate OSGiLarva property automata into NewMS property automata.

5.2.1 New property events from SSU layer

In the NewMS system, the SSU layer checks, at each service method call, whether the service reference is a stale reference or not. If the reference is stale, then the call is paused and the SSU layer automatically manages the stale reference with three possible cases: *Substitution*, *No alternative service* and *Roll Back failed*.

- *Substitution*: the client and other currently used services are restored in the same status as before the service providing the stale reference was used. The SSU layer performed a service substitution and a *Substitution* event is raised.
- *No alternative service*: the SSU layer didn't find an alternative service but the client roll back went well, a *SRException* event is sent and a stale reference exception is thrown to the client. In this case, the client must find a way to handle the missing service by itself.
- *Roll back failed*: the client is stocked in a previous roll back call, that means it is already in a stale reference management and a new one occurs concurrently. We do not want to handle this case. We then notify client that we are in trouble. A *RBException* event is sent to the monitor, prior to the raise of a roll back exception.

These three events are an approach to place the monitoring automata in the most accurate position when a stale reference occurs.

5.2.2 OSGiLarva translation to NewMS

The NewMS system gives an accurate life-cycle management in-line with OSGi operations. We design an algorithm (**Algorithm 1** shown in Fig. 5.2) to automatically translate an OSGiLarva's property automaton into a NewMS's property automaton. Since the three additional property events of NewMS are more precise than the framework events of OSGiLarva, we remove all transitions associated to the OSGi framework events, then add three transition functions associated to the new defined property events in NewMS's property automata. In this algorithm, some generated transitions are labelled by true and nop. They correspond to transitions always crossable without condition and without action.

The function $compose(l_1, l_2)$ (**Algorithm 2** shown in Fig. 5.3) is used to merge transitions starting from the same state and fired by the same event. Indeed, when a transition is introduced from s_1 to s_2 to step over a removed transition associated to a framework event, the new transition corresponds to an event that is already triggered

Data: Let A = (S, s_0, B, V, v_0, Σ_M, Σ_F, δ) an OSGiLarva automaton which will be updated to generate a NewMS automaton. Some elements will not be changed: s_0, V, v_0, Σ_M.

step 1. Keep only transitions associated to methods call events in δ of A:

while *for each* t_1 *and DS, and* $(t_1=(s, e_1, l_1) \wedge t_1 \in \delta \wedge e_1 \in \Sigma_F \wedge DS= \{ds \mid \exists (c, a) ((c, a, ds) \in l_1)\})$, **do**

  δ=δ-t_1;

  while *for each* s_2, $s_2 \in DS$, **do**

    if s_2 *is not reachable from the initial state* s_0, **then**

      while *for each* t_2 *starting from* s_2, $(t_2=(s_2, e_2, l_2) \wedge t_2 \in \delta)$, **do**

        generate new transition like t_2 but starting from s and add it in δ, **by:**

        if *a transition* t_3 *with* e_2 *already exists* $(t_3=(s, e_2, l_3) \wedge t_3 \in \delta)$, **then**

          $\delta = (\delta - \{t_3\}) \cup \{(s, e_2, compose(l_3; l_2))\}$;

        else if *no transition like* t_3, **then**

          $\delta = \delta \cup \{(s, e_2, l_2)\}$;

        end

      end

    end

  end

end

step 2. delete all framework events names from Σ_F and add the three new property events names in Σ_F: $\sum_F = \{SRException, RBException, Substitution\}$;

step 3. add two new bad states in S and B of A:

$S = S \cup \{RBExp, SRExp\}$;

$B = B \cup \{RBExp, SRExp\}$;

step 4. From all non bad state, if "RBException" event occurs, it reaches state RBExp.

$\delta = \delta \cup (S - B) \times \{RBException\} \times [(true, nop, RBExp)]$;

step 5. From all non bad state, if "SRException" event occurs, it reaches state SRExp.

$\delta = \delta \cup (S - B) \times \{SRException\} \times [(true, nop, SRExp)]$;

step 6. From all non bad state, if "Substitution" event occurs, it reaches the initial state s_0.

$\delta = \delta \cup (S - B) \times \{Substitution\} \times [(true, nop, s_0)]$;

Figure 5.2: Generate NewMS automata from OSGiLarva automata(**Algorithm 1**)

from the state s_1. In such a case, we propose to use a permissive merging heuristic based on a normal behavior first policy. It means that we give priority to behaviors reaching non-bad states. In this policy, we would try to respect following requirements:

1. keep the internal behavior (order) of each list;
2. the normal states of l_1 with its normal behaviors have priority over l_2's;
3. all l_1 bad states are pushed after the normal states of l_1;

The first step of this algorithm is inspired by the *bubble sort algorithm* to have normal states before bad ones of each list, but keeping their priority. The second step composes both lists together by a non-bad states part and bad states part.

These translated property automata can be implemented by OSGiLarva property description language to monitor OSGi-based systems. In the next section, we give an example to show the translation by the algorithms.

Data: l_1 and l_2 are two lists of transition labels, where each label is a 3-tuple composed by a condition, an action and a destination state,

step 1. push bad states at the end of each list, by keeping original behaviors with the following method on each list:

while *there exists a location i in the list l such as state* s_i *is a bad state and* $s_{(i+1)}$ *is not a bad state,* **do**

swap labels and enforce condition of the pulled label.
For instance, this list
$l=\{(c_0, a_0, s_0), \cdots, (c_i, a_i, s_i), (c_{(i+1)}, a_{(i+1)}, s_{(i+1)}), \cdots\}$,
is transformed into
$l=\{(c_0, a_0, s_0), \cdots, ((c_{(i+1)} \wedge \neg\, c_i), a_{(i+1)}, s_{(i+1)}), (c_i, a_i, s_i), \cdots\}$,

end

Step 2. merge both lists by keeping the following global structure:

compose(l_1,l_2)= {(sub-list of l_2 without bad states);
(sub-list of l_1 without bad states);
(sub-list of l_2 with bad states);
(sub-list of l_1 with bad states)}

end

Figure 5.3: Compose(l_1, l_2): composes two new lists of transitions (**Algorithm 2**)

5.2.3 Example of automata translation

In order to show the translation from an OSGiLarva automaton to a NewMS automaton, we present a translation example in this section. The Airline Reservation system (from Section 3.4.1.4) with OSGiLarva specification is an example be used at here. The **Algorithm 1** is applied to translate the OSGiLarva automaton A into a NewMS property automaton A' (Fig.5.4). Fig. 5.5 corresponds to the graphical representation of this translation.

During the check of an execution by our monitor, if a destination state of a transition is a bad state (i.e., states "error", "RBExp" and "SRExp" in Fig. 5.4 and Fig. 5.5), this trace is error ending. The identification of the faulty behavior is written in the NewMS log. If all destination states are non-bad states, this trace is correct with respect to the property.

On Fig. 5.4 and Fig. 5.5, we could observe that there is no any explosion of the size of the property, although we added some new events and new states. Indeed, the number of the added transitions increases linearly with the number of non-bad

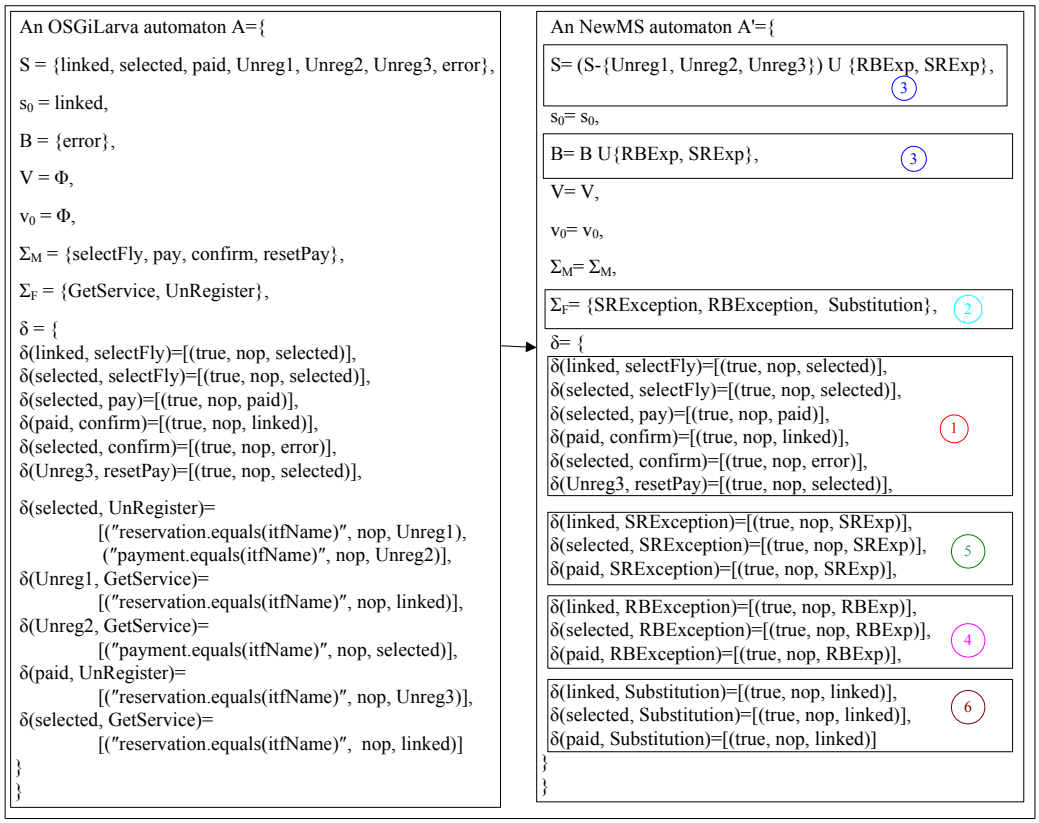


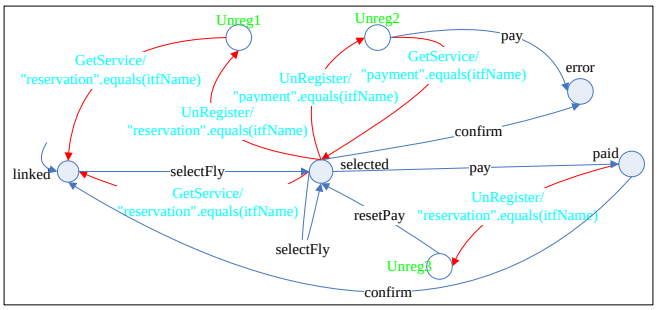
Figure 5.4: A translation example from an OSGiLarva automaton to NewMS automaton indicating algorithm steps

states in the property, and the size of the produced automaton is reduced because of the removal of framework transitions.

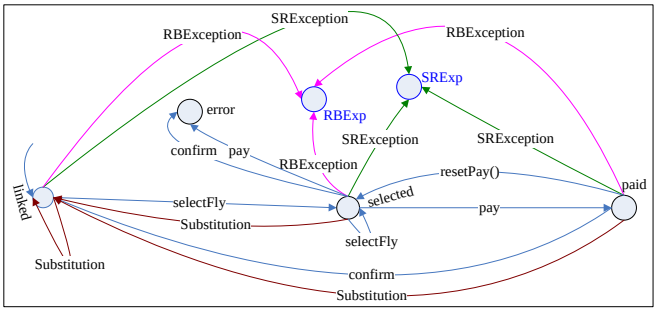
5.2.4 Expressiveness gains

The OSGiLarva can check a service unregistered, registered and loaded, but it is not sure whether the current used service reference is stale or not and whether the using services are coherent or not. Conversely, NewMS is guaranteeing all of them, since the SSU layer automatically manages services for clients. Moreover, the new cases (i.e., *Substitution*, *No alternative service* and *Roll Back failed*) which obtained by the SSU layer make properties to be more precise and allow to detect more details in NewMS than in OSGiLarva.

To illustrate the main benefit of NewMS, we propose to discuss about the execution trace t_2 which is the same as in Section 3.4.1.4. Its corresponding events trace with NewMS is $t''_2 = [selectFly, pay, resetPay, Substitution, selectedFly, pay, confirm]$.



a. An OSGiLarva automaton of the airline reservation property



b. A translated NewMS automaton of the airline reservation property

Figure 5.5: Translate an OSGiLarva automaton A to a NewMS automaton A'

Although t_2 is correct by OSGiLarva and NewMS system verifying, when OSGiLarva checks t_2' (in Section 3.4.1.4), a stale reference could be hidden in t_2. For instance, it could be the case that a client gets a service (S_2), but he still uses an old one (S_1) through a stale reference. However, t_2 can avoid stale references by the SSU layer of the NewMS. Since NewMS manages itself whether the current service (S_1) is unregistered and whether a new service (S_2) can substitute it (S_1), then the SSU layer guarantees that the second invocation of the method $selectFly$ is correctly executed.

5.3 Implementation–OSGiLarva-SSU++

The OSGiLarva-SSU++ system (Fig. 5.6) integrates the SSU layer and OSGiLarva systems and implements NewMS architecture. This implementation mainly focuses on the property verification system and the combined work of LogOs system and SSU layer working proxies.

In Section 5.2.2, we proposed an algorithm to generate NewMS property automata from OSGiLarva property automata. This generated NewMS property automata can be described in a property file by OSGiLarva property description language. Before

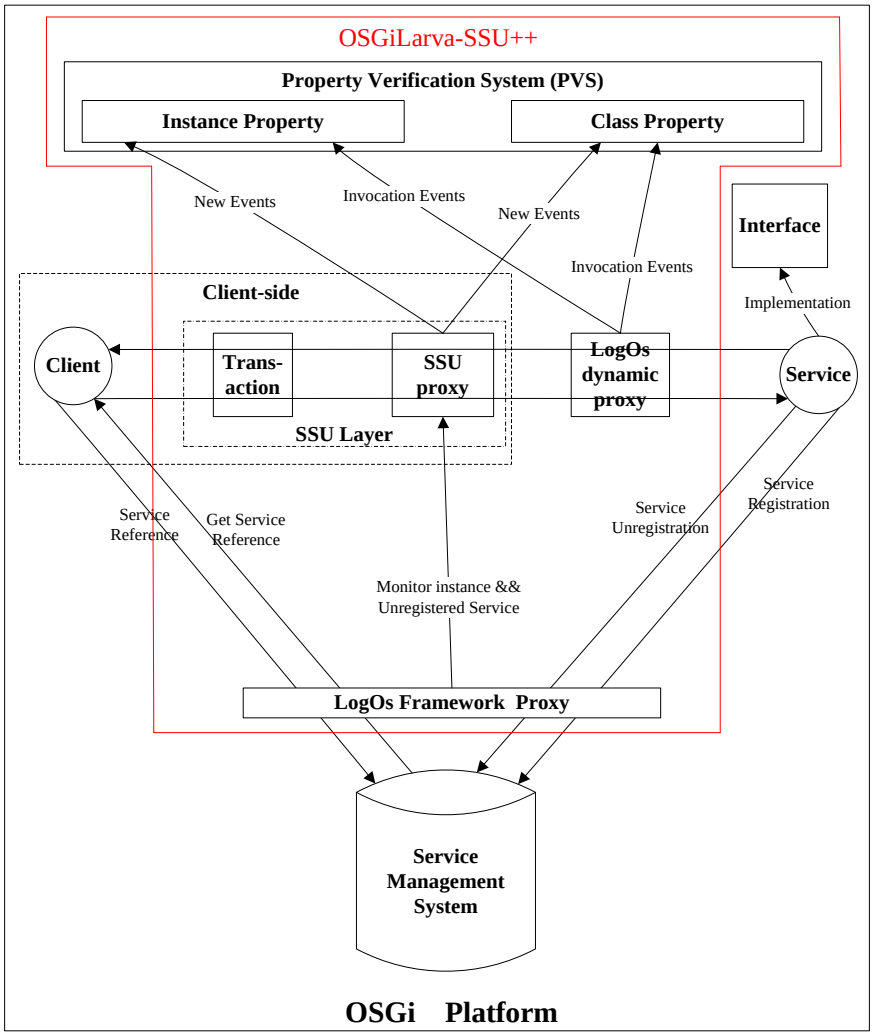


Figure 5.6: Implementation of the dynamic synthesized monitoring system

starting the monitored system, this NewMS property file must be compiled by our adapted Larva compiler to generate property verification system (PVS).

When client uses monitored services, the PVS starts to check the defined events which correspond to one or more method invocations from this client.

The client's method invocations are grouped in a *Transaction block*. The construction of this block automatically drives communications between client and server through *SSU proxy* and *LogOs proxy*. Since LogOs proxy is created at the time of service registering and SSU proxy is created before using the registered service, SSU proxy is created after LogOs proxy. SSU proxy is activated before LogOs proxy during clients use services. For example, when a client invokes method of a monitored service, its SSU proxy first intercepts this invocation.

Since SSU layer proxy gets an identification of the real service by referenced LogOs proxy and in order to avoid clients using stale reference in the SSU layer proxy, each time a service is unregistered, LogOs proxy notifies the existing SSU layers. So, when a service method call occurs, the SSU layer proxy first checks whether the using service is unregistered or not:

1. *Unregistered:* If the used service is unregistered, according to the policy, a substitution is tried and the corresponding new event (i.e., *Substitution* or *SRException* or *RBException*) is sent from its SSU proxy to the PVS.

2. *Registered:* If the using service is registered, SSU proxy handles this request through LogOs proxy, LogOs proxy directly requests it to server and this request is sent from LogOs proxy to the PVS.

Finally, the monitoring records are real-time sent to users from PVS.

5.4 Summary

In this chapter, we described the NewMS system. It combines the OSGiLarva system and the SSU layer. Main features are *dynamicity resilience*, *comprehensiveness* and *fault tolerance* inherited from these two systems. The integration can observe three possible cases when a unregistered service is being used, such as: *substitution*, *No alternative service* and *Roll back failed*. The NewMS can manage the three cases and check their correspond event through the described property automata.

We also design an algorithm to automatically translate OSGiLarva property automata into NewMS property automata.

Finally, comparing with OSGiLarva monitoring system, the NewMS system's main advantages are the accuracy of the interpretation to check whether a reference is stale or not.

6

Conclusions and Perspectives

In this chapter, we give our conclusions of this thesis and some perspectives.

6.1 Conclusions

In a dynamic SOA architecture for open environment, like OSGi, services are loosely coupled and their life-cycle management is dynamic. In this context, services may dynamically disappear and appear to make a substitution of service at runtime. There exists some monitors able to handle such services usage, but they have some constrains. It means these monitors disappear at service unregistered time. The new appeared service can't be checked by any monitor without restarting the monitored system, and stale references may be used by other services. In this thesis, considering the aforementioned problems, we proposed a dynamic monitoring system to check whether these behaviors of clients are authorized or not and a SSU layer to automatically handle the stale references at runtime.

In Chapter 2, we introduced the background of this thesis. Then, we reviewed the main existing monitoring systems and some approaches about service substitution. We mainly done a forcus on approaches handling stale references in dynamic SOA systems for open environment. As shown in these proposed approaches, there are

some limitations to dynamically monitor services while handling stale references.

Chapter 3 expressed the relationships between client and services in order to design a dynamic monitoring system. We defined two key principles of the dynamic monitoring system: *dynamicity resilience* and *comprehensiveness*. The former refers to some information of monitor (i.e., the current state and the monitor) which should be kept in memory while the monitored service is unregistered, if this service is substituted by another one, the stored information also needs to be transferred. It means that this monitor doesn't vanish with the disappeared service. The latter means that no service implementation of this service interface can bypass its monitor's observation. Besides, we also proposed two levels of property description: client-side property (*Instance-Property*) and interface-side property (*Class-property*). *Instance-Property* refers to each client's instance of the property is generated when this client uses monitored services. The life-cycle of the instance property is the same as its corresponding client. *Instance-properties* have the same property description except for client's id. Such properties descriptions are expressed in the context of a **foreachclient**. One *Class-Property* can express a centralized property for one interface to protect atomic usage of services. It is the entrance of the monitoring system. Its life-cycle is the same as its service interface rather than service implementation. So, it embodied the dynamicity resilience of our monitoring system. Moreover, OSGiLarva supports a property file expressed for multiple services interfaces. Finally, we established that OSGiLarva has equivalent performances for monitoring dynamic SOA systems as Larva tool.

For resolving the stale references issues from dynamic SOA systems, we designed a safe service usage (SSU) layer at client-side in Chapter 4. This layer aids clients to check references before using them and automatically deal with stale references without restarting clients and without any constraints on services. There are two situations to resolve stale references: either make a service substitution even if unregistered services are stateful services or throw a stale reference exception to clients. In order to keep coherent service usage after services substitution, we use a transactional approach to express the use of stateful services. Client can use one or more stateful services at same time. All the requests are expressed in a transaction block. When a used service is unregistered, the transaction block executes a roll-back and reverts all variables' values which are used in this transaction block, and then SSU layer copes with the stale reference for client. The results show that the time costs of original software system with or without SSU layer are close.

The two previous proposed approaches can monitor dynamic SOA systems and handle stale references at runtime. However, OSGiLarva isn't aware of stale references at runtime. Therefore, we merged both approaches into the NewMS tool at Chapter 5. It dynamically checks whether the behaviors of clients using services are

authorized or not, and it handles stale references without adding extra assumptions on services. NewMS has more extensive range of observation than the OSGiLarva system has. Indeed, it can describe more precisely property for the monitored services, since there are three additional events associated to the dynamicity of the monitored services. In order to describe NewMS property automata in a property file, we designed an algorithm to automatically generate NewMS property automata from the described OSGiLarva property automata. Finally, the implementation of this NewMS architecture is a work in progress.

6.2 Perspectives

Although the work in this thesis proposed monitoring systems supporting the dynamicity of SOA systems, the avoiding stale references, there are still several interesting directions which can be taken in future:

1. **Non-hard-coding** Although our latest approach of monitoring system given in Chapter 5 avoids stale reference usage, there is a restriction according to OSGiLarva: in order to protect coherent services use with transactional approach, the implementation of NewMS architecture has to change the code of clients in the monitored software system. In the future, we aim at providing such transactional approach in property description as special constructs which check the service usage and without changing the client code.

2. **Partial-stateful \ Partial-stateless** In this thesis, we just consider our NewMS system with stateful feature to monitor the stateful communications in dynamic SOA systems. In order to reduce the cost overhead, we can add some special annotations in service interfaces of the monitored system so that the monitoring system identify it and give some corresponding operations to monitor, such as, *stateless*, *stateful*. For example, if a being used service is unregistered, this monitoring system need to check and reset its related services the interfaces of whom is annotated by *stateful*, and just keep the current state of the related services whose interfaces is annotated by *stateless*. Given that such a partial-stateful and partial-stateless monitoring system is used in a real-life scenarios, it would be an interesting proposition.

3. **Efficiency** Since we proposed monitoring systems synchronized with the monitored software system, it induced some time cost. In a next version of the tool, we could make some propositions to reduce our tools time cost. For instance, we could make our monitoring systems asynchronous, by exporting monitors

to separate threads, or we can limit monitoring to only occur within a fixed period of time: if the property is respected during one week by a given consumer, we can consider that it will still respect it afterwards.

4. **Property to test cases** Although we have formally specified our monitoring property description to verify the given clients behaviors, we have not yet carried out an in depth analysis with our monitoring system to monitor a large number of different clients behaviors. It is very time-consuming to manually write a large quantity of test cases for different clients behaviors. The Tobias tool [68] can generate a large number of repetitive test cases through all combinations of given parameter values in a simple text file. We assume that we can automatically generate all possibilities clients' invocations according to the events described in the property file. All the generated clients' invocation orders is a set of test suites of clients' behaviors which may occur in its system. This significantly reduce time-consuming in the test phase. Since the combination elements are the specified monitored events, the way has comprehensiveness of test suites generation.

5. **Industrial application** While we have proposed the theory and implementations of our monitor systems and applied on industry-inspired case studies, this monitor-oriented dynamicity programming through OSGiLarva property automata or NewMS property automata have not been tried out on industrial-scale case studies. Indeed such applications to real-life scenarios would be vital for transforming the current prototype SOA-based systems into a mature and reliable one.

References

[1] Open Service Gateway Initiative (OSGi), http://www.osgi.org/ [retrieved: June, 2012].

[2] Javascript object notation web service protocol. http://en.wikipedia.org/wiki/JSON-WSP.

[3] Web service description language: Wsdl 1.1, 2001.

[4] Simple object access protocol:SOAP 1.2. W3C, 2003.

[5] Transactional systems. In *Reliable Distributed Systems*, pages 509–528. Springer New York, 2005.

[6] An autonomic approach to offer services in OSGi-based home gateways. *Computer Communications*, 31(13):3049 – 3058, 2008. Special Issue:Self-organization and self-management in communications as applied to autonomic networks.

[7] H. Ahn, H. Oh, and J. Hong. Towards Reliable OSGi Operating Framework and Applications. *Journal of Information Science and Engineering*, 23(5):1379, 2007.

[8] OSGi Alliance. About the OSGi service platform. Technical report, June 7 2007.

[9] The OSGi Alliance. OSGi service platform core specification. Revision 4.1, April 2007.

[10] Sven Apel and D. Batory. How AspectJ is used: an analysis of eleven AspectJ programs. *Journal of Object Technology*, 9:117–142, 2010.

[11] A. Aviziens. Fault-tolerant systems. *IEEE Trans. Comput.*, 25(12):1304–1312, December 1976.

[12] Stefan Axelsson, Ulf Lindqvist, and Ulf Gustafson. An approach to UNIX security logging. In *21st National Information Systems Security Conference*, pages 62–75, 1998.

[13] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '06, pages 63–71. IEEE Computer Society, 2006.

[14] Luciano Baresi, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini. Validation of web service compositions. *IET Software*, 1(6):219–232, 2007.

[15] Mike Barnett, Robert DeLine, Manuel Fahndrich, Bart Jacobs, K.RustanM. Leino, Wolfram Schulte, and Herman Venter. The spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer Berlin Heidelberg, 2008.

[16] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: an overview. In *Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[17] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Reconfiguration Service for CORBA. *International Conference of Configurable Distributed Systems*, 1998.

[18] JanOlaf Blech, Yliès Falcone, Harald Rueß, and Bernhard Schätz. Behavioral specification based runtime monitors for OSGi services. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 405–419. Springer Berlin Heidelberg, 2012.

[19] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.

[20] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480, April 2008.

[21] Humberto Cervantes and Richard S. Hall. Automating Service Dependency Management in a Service-Oriented Component Model. In *CBSE*, 2003.

[22] Patrice Chalin, JosephR. Kiniry, GaryT. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and esc/java2. In

FrankS. Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer Berlin Heidelberg, 2006.

[23] Mala chandra. Seamless mobility and OSGi service platform. Technical report, Client Application and Architecture Motorola, Inc., 2004.

[24] Feng Chen. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Electronic Notes in Theoretical Computer Science*, pages 106–125. Elsevier Science, 2003.

[25] Feng Chen and Grigore Rosu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer Berlin Heidelberg, 2005.

[26] Hao-Chung Cheng, Wei-Tsong Lee, Xin-Wen Wei, and Tian-Wen Sun. A novel service oriented architecture combined with cloud computing based on R-OSGi. In James J. (Jong Hyuk) Park, Qun Jin, Martin Sang-soo Yeo, and Bin Hu, editors, *Human Centric Technology and Service in Smart Space*, volume 182 of *Lecture Notes in Electrical Engineering*, pages 291–296. Springer Netherlands, 2012.

[27] Yoonsik Cheon, Yoonsik Cheon, Gary T. Leavens, and Gary T. Leavens. A runtime assertion checker for the java modeling language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP), Las Vegas*, pages 322–328. CSREA Press, 2002.

[28] Yoonsik Cheon, Gary T. Leavens, Yoonsik Cheon, and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002, volume 2374 of LNCS*, pages 231–255. Springer Berlin Heidelberg, 2002.

[29] Hsin Chou. Service oriented architecture for an overall radioactive waste package record management system. *Progress in Nuclear Energy*, 53:420–427, May 2011.

[30] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV'08, pages 121–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS*, pages 135–149, 2008.

[32] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva - safer monitoring of real-time java programs. In *SEFM*, 2009.

[33] Christian Colombo, GordonJ. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer Berlin Heidelberg, 2009.

[34] Yufang Dan, Nicolas Stouls, Christian Colombo, and Stéphane Frénot. OSGi-Larva: a monitoring framework supporting OSGi's dynamicity.

[35] Yufang Dan, Nicolas Stouls, Stéphane Frénot, and Christian Colombo. A Monitoring Approach for Dynamic Service-Oriented Architecture Systems. In *SERVICE COMPUTATION 2012: The Fourth International Conferences on Advanced Service Computing*, pages 20–23, 2012.

[36] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Zohar Manna. Lola: runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on*, pages 166–174, 2005.

[37] Rogerio de Lemos, Paulo Asterio de Castro Guerra, and Cecilia Mary Fischer Rubira. A Fault-Tolerant Architectural Approach for Dependable Systems. *IEEE Software*, 23:80–87, 2006.

[38] Andre de Matos Pedro. *Dynamic contracts for verification and enforcement of real-time systems properties*. PhD thesis, Cister Research Unit - ISEP/IPP, September 17 2012.

[39] Dionysis Athanasopoulos, Apostolos Zarras, and Valérie Issarny. Service Substitution Revisited. In *24th IEEE/ACM International Conference on Automated Software Engineering - ASE 2009*, Auckland Nouvelle-Zélande, 11 2009. IEEE/ACM.

[40] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, PÅl Krogdahl, Dr Min Luo, and Tony Newling. *Patterns: Service-Oriented Architecture and Web Services*. Number SG24-6303-00. IBM Redbooks, ibm edition, April 29 2004.

[41] Paul England. Practical Techniques for Operating System Attestation. In *1st international conference on Trusted Computing and Trust in Information Technologies (Trust'08)*, pages 1–13. Springer-Verlag, 2008.

[42] Clement Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In *Services Computing, IEEE International Conference on*, pages 474–481. IEEE Computer Society, 2007.

[43] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[44] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Notices*, 37(5):234–245, May 2002.

[45] The Eclipse Foundation. http://eclipse.org/aspectj/, 2013.

[46] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras. Dynamic service substitution in service-oriented architectures. In *Services - Part I, 2008. IEEE Congress on*, pages 101–104, July 2008.

[47] Stéphane Frénot and Julien Ponge. LogOS: an Automatic Logging Framework for Service-Oriented Architectures. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Izmir, Turquie, September 2012.

[48] Kiev Gama and Didier Donsez. Service Coroner: A Diagnostic Tool for Locating OSGi Stale References. In *34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA*, pages 108–115. IEEE, 2008.

[49] Kirupa Ganapathy, Bharathi Priya, Bhanu Priya, Dhivya, V. Prashanth, and V. Vaidehi. SOA framework for geriatric remote health care using wireless sensor network. *Procedia Computer Science*, 19(0):1012 – 1019, 2013. The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013), the 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013).

[50] Selvin George, David Evans, and Steven Marchette. A biological programming model for self-healing. In *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems: in association with 10th ACM Conference on Computer and Communications Security*, SSRS '03, pages 72–81, New York, NY, USA, 2003. ACM.

[51] D Ghosh, R Sharman, H Raghavrao, and S Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[52] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification (The 3rd Edition)*. Addison-Wesley Professional, 2005.

[53] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 321–336, Berlin, Heidelberg, 2009. Springer-Verlag.

[54] Hugo Haas and Allen Brown. Web services glossary. W3C, February 11 2004.

[55] Helen Hawkins and Ron Bodkin. Aspectj. The Eclipse Foundation, 2013.

[56] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[57] Vince Izzo. Motorola connected home experience with OSGi. Technical report, Motorola Inc., 2006.

[58] Thierry Jeron, Herve Marchand, Antoine Rollet, Ylies Falcone, and Omer Nguena Timo. Runtime Enforcement of Timed Properties. In *3rd international conference on Runtime Verification (RV)*, Septembre 2012.

[59] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. Javamop: Efficient parametric runtime monitoring framework. In *Proceeding of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1427–1430. IEEE, 2012.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[61] Young-Gab Kim, Chang-Joo Moon, Dae-Ha Park, and Doo-Kwon Baik. A mac-based service bundle authentication mechanism in the OSGi service platform. In YoonJoon Lee, Jianzhong Li, Kyu-Young Whang, and Doheon Lee, editors, *Database Systems for Advanced Applications*, volume 2973 of *Lecture Notes in Computer Science*, pages 137–145. Springer Berlin Heidelberg, 2004.

[62] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[63] Chin-Feng Lai, Yi-Wei Ma, Sung-Yen Chang, Han-Chieh Chao, and Yueh-Min Huang. OSGi-based services architecture for cyber-physical home control systems. *Computer Communications*, 34(2):184 – 191, 2011. Special Issue: Open network service technologies and applications.

[64] Daniel Le Métayer, Manuel Maarek, Eduardo Mazza, Marie-Laure Potet, Stéphane Frénot, Valérie Viet Triem Tong, Nicolas Craipeau, Ronan Hardouin, Christophe Alleaune, Valérie-Laure Benabou, Denis Beras, Christophe Bidan, Gregor Goessler, Julien Le Clainche, Ludovic Mé, and Sylvain Steer. Liability in Software Engineering Overview of the LISE Approach and Illustration on a Case Study. In *ICSE'10*, page 135. ACM/IEEE, 2010.

[65] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[66] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in java. In *IN OOPSLA 2000 COMPANION*, pages 105–106. ACM, 2000.

[67] GaryT. Leavens, AlbertL. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, volume 523 of *The Springer International Series in Engineering and Computer Science*, pages 175–188. Springer US, 1999.

[68] Y. Ledra and L. du Bousquet. Tobias-z: An executable formal specification of a test generator. In *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 353–354, Sept 2006.

[69] Rogerio de Lemos, Paulo Asterio de Castro Guerra, and Cecilia Mary Fischer Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Softw.*, 23(2):80–87, March 2006.

[70] Yossi Lev and Jan willem Maessen. Towards a safer interaction with transactional memory by tracking object visibility. In *In: SCOOL â05, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.

[71] Chao lin Wu, Chun feng Liao, and Li chen Fu. Service-oriented smart home architecture based on OSGi and mobile agent technology, June 2007.

[72] Zaigham Mahmood. Service oriented architecture: a new paradigm for enterprise application integration. In *Proceedings of the 11th WSEAS International Conference on Computers*, ICCOMP'07, pages 491–496, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[73] Chaudhry Maria, Ali Hammad Akbar, Ahmad Qanita, and Sarwar Imran. SOARware1: Treading through the crossroads of RFID middleware and SOA

paradigm. *Journal of network and computer applications*, 34, no 3 (235 p.):17(998–1014), 2011.

[74] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel, 1991.

[75] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An Overview of the MOP Runtime Verification Framework. *International Journal on Software Techniques for Technology Transfer*, 2011. Springer.

[76] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[77] Radhika Nagpal, Attila Kondacs, and et al. Programming methodology for biologically-inspired self-assembling systems, 2002.

[78] Choon-Sung Nam, Sukhan Lee, and Dong-Ryeol Shin. An android remote call vehicle service for OSGi-based unmanned vehicle using by a mobile device. In Sukhan Lee, Hyungsuck Cho, Kwang-Joon Yoon, and Jangmyung Lee, editors, *Intelligent Autonomous Systems 12*, volume 193 of *Advances in Intelligent Systems and Computing*, pages 123–132. Springer Berlin Heidelberg, 2013.

[79] Hamid R. Motahari Nezhad, Régis Saint-Paul, Fabio Casati, and Boualem Benatallah. Event correlation for process discovery from web service interaction logs. *VLDB J.*, 20(3):417–444, 2011.

[80] Dave Patterson and Armando Fox, editors. *Recovery-Oriented Computing*, 2004.

[81] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. big web services: Making the right architectural decision. In *17th International World Wide Web Conference (WWW2008)*, pages 805–814, Beijing, China, April 2008 2008.

[82] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: a hard real-time runtime monitor. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 345–359, Berlin, Heidelberg, 2010. Springer-Verlag.

[83] Alexandre Polozoff. Proactive application monitoring. Technical report, IBM Software Group, Software Services for WebSphere, Chicago, Illinois, April 2003.

[84] M. Psiuk. Aop-based monitoring instrumentation of jbi-compliant esb. In *Services - I, 2009 World Conference on*, pages 570–577, 2009.

[85] Marcus J. Ranum. System logging and log analysis. Technical report, 2004.

[86] Clifford S. Russell, editor. *Monitoring and enforcement*, 1885.

[87] C. Simache, M. Kaaniche, and A. Saidane. Event log based dependability analysis of windows nt and 2k systems. In *International Symposium on Dependable Computing*, pages 311–315, 2002.

[88] S.E. Smaha. Haystack: an intrusion detection system. In *Aerospace Computer Security Applications Conference, 1988., Fourth*, pages 37–44, 1988.

[89] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach for web services substitution. In *Database Engineering and Applications Symposium, 2006. IDEAS '06. 10th International*, pages 166–173, dec 2006.

[90] Herman Mekontso Tchinda, Julien Ponge, Yufang Dan, and Nicolas Stouls. An API for Autonomous and Client-side Service Substitution. In *SERVICE COMPUTATION 2012: The Fourth International Conferences on Advanced Service Computing*, pages 14–19, 2012.

[91] The OSGi Alliance. OSGi Services Platform, Core Specification, Version 4.2, June 2009. http://www.osgi.org [retrieved: June, 2012].

[92] Gael Thomas. La plateforme dynamique de service OSGi. Technical report, University Joseph Fourier PolyTech, Grenoble LIG/ADELE, Janvier 2007.

[93] Apache TomEE. Security annotations. http://openejb.apache.org/transaction-annotations.html, 2013.

[94] R. Varadan, K. Channabasavaiah, S. Simpson, K. Holley, and A. Allam. Increasing business flexibility and SOA adoption through effective SOA governance. *IBM Systems Journal*, 47(3):473–488, 2008.

[95] M. Vijay and R. Mittal. Algorithm-based fault tolerance: a review. *Microprocessors and Microsystems*, 21(3):151 – 161, 1997. Fault Tolerant Computing.

[96] Pierpaolo Vittorini, Monica Michetti, and Ferdinando di Orio. A SOA statistical engine for biomedical data. *Comput. Methods Prog. Biomed.*, 92(1):144–153, October 2008.

[97] Guanhua Wang. Improving data transmission in web applications via the translation between xml and json. In *Communications and Mobile Computing (CMC), 2011 Third International Conference on*, pages 182–185, April 2011.

[98] Ian Warren and Ian Sommerville. A model for Dynamic Configuration which Preserves Application Integrity. In *3rd ICCDS*. IEEE Computer Society, 1996.

[99] Ye-Chi Wu and Hewijin Christine Jiau. A monitoring mechanism to support agility in service-based application evolution. *SIGSOFT Softw. Eng. Notes*, 37(5):1–10, September 2012.

[100] Chu-Sing Yang, Ming-Yi Liao, and Chao xing Chen. Design and implementation of hems based on rfid and OSGi. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, pages 250–253, 2009.

[101] Hua Yue and Xu Tao. Web services security problem in service-oriented architecture. *Physics Procedia*, 24, Part C(0):1635 – 1641, 2012. International Conference on Applied Physics and Industrial Engineering 2012.

[102] Daniel zmuda, Marek Psiuk, and Krzysztof Zielinski. Dynamic monitoring framework for the SOA execution environment. *Procedia Computer Science*, 1(1):125 – 133, 2010. ICCS 2010.

[103] Marta Zorrilla and Diego Garcia-Saiz. A service oriented architecture to provide data mining services for non-expert data miners. *Decision Support Systems*, 55(1):399 – 411, 2013.

List of publications

Journal

[34] Yufang Dan, Nicolas Stouls, Christian Colombo and Stéphane Frénot. *OSGiLarva: a Monitoring Framework Supporting OSGi's Dynamicity*. International journal on advances in security, IARIA, 2013, 6 (1 and 2), pages. 49–61

International Conferences

[35] Yufang Dan, Nicolas Stouls, Stéphane Frénot and Christian Colombo. *A Monitoring Approach for Dynamic Service-Oriented Architecture Systems*. The Fourth International Conferences on Advanced Service Computing, SERVICE COMPUTATION 2012, pages 20–23. ISBN 978-1-61208-215-8. [Best Paper Award]

[90] Herman Mekontso Tchinda, Julien Ponge, Yufang Dan et Nicolas Stouls. *An API for Autonomous and Client-side Service Substitution*. The Fourth International Conferences on Advanced Service Computing, SERVICE COMPUTATION 2012, pages 14–19, ISBN 978-1-61208-215-8.

National Workshop

[104] Yufang Dan, Nicolas Stouls and Stéphane Frénot. *OSGiMOP: Monitoring-Oriented Programming in OSGi*. Journées scientifiques 2011 du projet SEmba, Valence, France.

[105] Yufang Dan, Nicolas Stouls, Christian Colombo and Stéphane Frénot. *OSGiLarva tool: a Monitoring Tool Supporting OSGi's Dynamicity*. Journées scientifiques 2013 du projet SEmba, Domaine des Hautannes, Saint Germain au Mont d'Or, France.