

University of Crete
Computer Science Department

Monitoring the QoS of Web Services using SLAs -
Computing metrics for composed services

Chrysostomos Zeginis
Master's Thesis

Heraklion, March 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

**Παρακολούθηση των QoS χαρακτηριστικών των ηλεκτρονικών υπηρεσιών
χρησιμοποιώντας SLA έγγραφα - Υπολογισμός μετρικών σε σύνθετες υπηρεσίες**

Εργασία που υποβλήθηκε από τον
Χρυσόστομο Κ. Ζεγκίνη
ως μερική εκπλήρωση για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Χρυσόστομος Ζεγκίνης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Δημήτρης Πλεξουσάκης, Καθηγητής, Επόπτης

Ευάγγελος Μαρκάτος, Καθηγητής, Μέλος

Ιωάννης Τζιτζικας, Επίκουρος Καθηγητής, Μέλος

Δεκτή:

Πάνος Τραχανιάς, Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών
Ηράκλειο, Μάρτιος 2009

Monitoring the QoS of Web Services using SLAs - Computing metrics for composed services

Chrysostomos Zeginis

Master's Thesis

Computer Science Department, University of Crete

Abstract

The Web services are an emerging technology which attracts a lot of attention from both academic and industry areas these recent years. Once Web services become operational, their execution needs to be managed and monitored to gain a clear view of how services perform within their operational environment, make management decisions and perform control actions to modify and adjust their behavior. Many approaches have been proposed and have proven that Web services' monitoring is very crucial for successful invocations.

This thesis analyzes the methods that can be used to monitor the execution of a Web service, in order to comply with the corresponding Service Level Agreement (SLAs) and the ways various metrics can be computed for composed services, having available the metric values of the constituent Web services. Nowadays, most of Web service providers sign SLA contracts with their clients to guarantee the offered functionality of their services. A monitoring system is introduced to check violations in the pre-agreed metric values of SLAs. These results can be very helpful for service providers, who can then take corrective actions to improve their services.

As far as the computation of QoS for composed Web services is concerned, this thesis proposes some basic metric types, provided that we know the composition pattern. Based on this pattern, appropriate formulas are used to compute the response time, the throughput, the reliability and the cost of the composed Web services. The validity of these formulas is verified with experimental results.

In summary, the contribution of this work lies in introducing a monitoring system, intended to check the Web services' compliance with SLAs, as well as in computing metrics for composed Web services.

Supervisor: Dimitris Plexousakis
Professor

Παρακολούθηση των QoS χαρακτηριστικών των ηλεκτρονικών υπηρεσιών χρησιμοποιώντας SLA έγγραφα- Υπολογισμός μετρικών σε σύνθετες υπηρεσίες

Χρυσόστομος Ζεγκίνης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Οι ηλεκτρονικές υπηρεσίες είναι μια αναπτυσσόμενη τεχνολογία που τραβάει όλο και περισσότερο την προσοχή τόσο της ακαδημαϊκής όσο και της βιομηχανικής κοινότητας. Από τη στιγμή που οι ηλεκτρονικές υπηρεσίες απέκτησαν λειτουργικότητα, η εκτέλεση τους πρέπει να ελέγχεται και να παρακολουθείται, έτσι ώστε να αποκτήσουμε μια ξεκάθαρη όψη πώς οι υπηρεσίες αποδίδουν μέσα στο λειτουργικό τους περιβάλλον, για να πάρουμε αποφάσεις διαχείρισής τους και για να εφαρμόσουμε πράξεις ελέγχου που έχουν ως σκοπό την τροποποίηση και την προσαρμογή της συμπεριφοράς τους. Πολλές προσεγγίσεις έχουν προταθεί γύρω από αυτό το θέμα και έχουν αποδείξει τη σημαντικότητά του για επιτυχημένη εκτέλεση ηλεκτρονικών υπηρεσιών.

Η συγκεκριμένη εργασία αναλύει τις μεθόδους, με τις οποίες μπορούμε να παρακολουθήσουμε την εκτέλεση μιας ηλεκτρονικής υπηρεσίας, ώστε να συμμορφώνεται με το αντίστοιχο SLA και τους τρόπους με τους οποίους μπορούμε να υπολογίσουμε διάφορες μετρικές για σύνθετες υπηρεσίες, έχοντας διαθέσιμες τις τιμές των μετρικών των υπηρεσιών που τις αποτελούν. Στις μέρες μας, σχεδόν όλοι οι πάροχοι ηλεκτρονικών υπηρεσιών υπογράφουν συμβόλαια SLA με τους πελάτες τους για να εγγυηθούν την προσφερόμενη ποιότητα των υπηρεσιών τους. Στην εργασία αυτή προτείνεται ένα σύστημα που ελέγχει για πιθανές αποκλίσεις στις συμφωνηθέντες τιμές των μετρικών. Στη συνέχεια, αυτά τα αποτελέσματα μπορεί να είναι πολύ χρήσιμα για τους πάροχους, που μπορούν με διορθωτικές ενέργειες να βελτιώσουν τις υπηρεσίες τους.

Όσον αφορά την μέτρηση της ποιότητας των σύνθετων υπηρεσιών, αυτή η εργασία προτείνει κάποιους βασικούς τύπους μετρικών, με την προϋπόθεση ότι γνωρίζουμε το πρότυπο της σύνθεσης. Βασισμένοι σε αυτό, μπορούμε να υπολογίσουμε το χρόνο απόκρισης, το ρυθμό εξυπηρέτησης, την αξιοπιστία και το κόστος των σύνθετων υπηρεσιών. Η εγκυρότητα των αποτελεσμάτων αποδεικνύεται μέσα από πειραματική μελέτη.

Συμπερασματικά, η συνεισφορά αυτής της εργασίας έγκειται στην παρουσίαση ενός συστήματος παρακολούθησης, που έχει ως σκοπό τον έλεγχο συμμόρφωσης της ηλεκτρονικής υπηρεσίας με το SLA, καθώς επίσης και στον υπολογισμό μετρικών για σύνθετες υπηρεσίες.

Επόπτης Καθηγητής: Δημήτρης Πλεξουσάκης
Καθηγητής

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη καθηγητή μου κ. Δημήτρη Πλεξουσάκη για την άπογη συνεργασία μας τα τελευταία 2 χρόνια, καθώς επίσης και για την ουσιαστική του καθοδήγηση και συμβολή στην ολοκλήρωση της παρούσας εργασίας.

Επίσης, θα ήθελα να εκφράσω τις ευχαριστίες μου, στους καθηγητές κ. Ευάγγελο Μαρκάτο και κ. Ιωάννη Τζιτζικά, για τη μεγάλη προθυμία τους να συμμετέχουν στην τριμελή εξεταστική επιτροπή.

Παράλληλα, θα ήθελα να ευχαριστήσω τον Κυριάκο Κρητικό για τη μεγάλη του βοήθεια και καθοδήγηση σε όλη τη διάρκεια εκπόνησης της εργασίας.

Πολλές ευχαριστίες θα ήθελα να εκφράσω στους φίλους μου, Αγάπη Περυσινάκη, Ελευθερία και Ελένη Σπυριδάκη, Κάλλια Μπαθιανάκη, Θέμη Δακανάλη, Κώστα Καστελλάκη, Βαγγέλη Μάγγα και Αντώνη Παπαδογιαννάκη για τη στήριξή τους και για όλες τις στιγμές που μοιραστήκαμε μαζί όλο αυτό τον καιρό.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαιτέρως τους γονείς μου, Κωνσταντίνο και Ελένη και τον αδερφό μου, Δημήτρη, για την υποστήριξη και την αγάπη με την οποία με περιέβαλλαν.

Contents

List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Web services	1
1.1.1 History of Web services	2
1.1.2 Types of Web services	3
1.1.3 Differences between Web services and Web pages	4
1.2 The Web services technology stack	5
1.3 The need for monitoring	8
1.4 Quality of service (QoS)	10
1.5 Service-Level Agreements (SLAs)	12
1.5.1 Why a Service Level Agreement is Important	15
1.5.2 SLA Life Cycle	15
1.5.3 Qualities That Can Be Defined in an SLA	17
1.6 Contributions	19
1.7 Organization of the thesis	19
2 Monitoring of Web services	20
2.1 Introduction	20
2.2 Services involved in compliance monitoring	22
2.3 Monitoring and Managing SLAs	24
2.3.1 Why we use WSLA	27
2.3.2 SLA Parameters and metrics	31
2.3.3 Obligations	33
2.3.4 Service level objectives	33
2.3.5 Action guarantees	35
2.4 Implementation	37
2.4.1 Implementing Tools	37
2.4.2 Implementation procedure	41
2.4.3 UML diagrams	45
3 Computing QoS Values of WS Compositions	48
3.1 Introduction	48
3.2 Types of Web service composition	49
3.2.1 Static Services Composition	49
3.2.2 Dynamic Services Composition	51

3.3	Web service composition patterns	51
3.3.1	Sequence Pattern	51
3.3.2	Parallel pattern	52
3.3.3	Synchronization pattern	52
3.3.4	Exclusive choice pattern	52
3.3.5	Simple merge pattern	53
3.3.6	Conditional pattern	53
3.3.7	Synchronizing merge pattern	54
3.3.8	Multi-merge pattern	54
3.3.9	Loop pattern	55
3.3.10	Deferred choice pattern	55
3.3.11	Interleaved Parallel Routing	55
3.3.12	Milestone pattern	56
3.4	Computing basic metrics for composed services	57
3.4.1	Computing response time	57
3.4.2	Computing throughput	58
3.4.3	Computing reliability	60
3.4.4	Computing cost	61
3.5	Summarizing the results	62
3.6	Use case	63
4	Experimental Evaluation	69
4.1	Introduction	69
4.2	Experiments	69
5	Related Work	76
5.1	Monitoring Approaches	76
5.1.1	Monitoring Web service compositions	76
5.1.2	Assertion-Based monitoring	80
5.1.3	Run-Time monitoring	82
5.1.4	Planning and monitoring service requests	87
5.1.5	Monitoring tools	88
5.2	Comparing monitoring approaches	92
5.3	Computing metrics of composed services	94
5.4	Comparison of our work with existing studies	96
6	Conclusion and Future Work	98
	Bibliography	100

List of Tables

Table 2.1: Assessment of the introduced approaches	30
Table 3.1: Metric Types for composed Web services.....	62
Table 3.2: Metric values for constituent Web services (use case).....	64
Table 4.1: Response time for the constituent and the composed Web service	72
Table 4.2: Throughput for the constituent and the composed Web service.....	73
Table 4.3: Response Time and throughput	74
Table 5.1: Comparing monitoring approaches.....	94

List of Figures

Figure 1.1: The Web Service technology stack.....	6
Figure 1.2: SLA Life Cycle.....	16
Figure 2.1: Services involved in SLA-compliance monitoring with multiple parties	23
Figure 2.2: Conceptual Architecture for SLA Monitoring and Management	25
Figure 2.3: SLA structure as defined in WSLA	31
Figure 2.4: Sample Elements of Service Description	32
Figure 2.5: Metric example in WSLA.....	32
Figure 2.6: SLA parameter example in WSLA.....	33
Figure 2.7: Service Level Objective example in WSLA.....	35
Figure 2.8: Action Guarantee example in WSLA	36
Figure 2.9: Main window of the WSLA plug-in for eclipse	39
Figure 2.10: A sample WSLA document extracted from the eclipse plug-in	40
Figure 2.11: WSLA elements stored in the PostgreSQL database.....	42
Figure 2.12: Metrics values in asadmin console	43
Figure 2.13: Graphical depiction of the Response Time.....	43
Figure 2.14: The UML class diagram of our implementation.....	45
Figure 2.15: The UML sequence diagram	47
Figure 3.1: Orchestration and choreography	50
Figure 3.2: Sequence pattern.....	52
Figure 3.3: Parallel pattern	52
Figure 3.4: Synchronization pattern	52
Figure 3.5: Exclusive choice pattern	53
Figure 3.6: Simple merge pattern	53
Figure 3.7: Conditional pattern	53
Figure 3.8: Synchronizing merge pattern	54
Figure 3.9: Multi-merge pattern	54
Figure 3.10: Loop pattern.....	55
Figure 3.11: Deferred choice pattern.....	55
Figure 3.12: Interleaved parallel routing.....	56
Figure 3.13: Milestone pattern	56
Figure 3.14: UML activity diagram for use case	63
Figure 3.15: Response time activity diagram (use case).....	64
Figure 3.16: Throughput activity diagram (use case)	65
Figure 3.17: Reliability activity diagram (use case)	67
Figure 3.18: Cost activity diagram (use case).....	68
Figure 4.1: Response Time and number of BPEL activities	70
Figure 4.2: Response time and number of requests	71

Figure 4.3: Response Time of composed Web services	72
Figure 4.4: Throughput of composed Web services	73
Figure 4.5: Relationship between response time and throughput	74
Figure 5.1: The Active BPEL engine extended with the run-time monitor environment	77
Figure 5.2: The domain monitor generation algorithm.....	79
Figure 5.3: A standard process annotated with contracts transformed into a monitored.....	81
Figure 5.4: Monitoring framework	82
Figure 5.5: Transforming the ECA rules to Statecharts.....	84
Figure 5.6: WS-Policy definitions and attachments.....	86
Figure 5.7: BP-Mon architecture	91
Figure 5.8: The travel site Web service	94
Figure 5.9: Pseudo code for computing QoS of a WS-workflow	96

Chapter 1

1 Introduction

1.1 Web services

A Web Service is a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application. Web services constitute a distributed computer infrastructure, made up of many different interacting application modules trying to communicate over private or public networks to virtually form a single logical system.

A more precise definition is published by the World Wide Web Consortium (W3C):

A web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols. [34]

In essence, a web service provides an interface defined in terms of XML messages and that can be accessed over the Internet (or, of course, an Intranet). It is an application that exposes a function which is accessible using standard Web technology and that adheres to Web services standards. This is significant because Web services are developed for and deployed onto any platform using any programming language.

A Web service can be: (i) a self-contained business task, such as funds withdrawal or funds deposit service; (ii) a full-fledged business process, such as the automated purchasing of office supplies; (iii) an application, such as a life insurance application or demand forecasts and stock replenishment; or (iv) a service-enabled resource, such as access to a particular back-end database containing patient medical records. Web services can vary in function from simple requests (e.g. credit checking and authorization, pricing enquiries, inventory status checking, or a weather report) to complete business applications that access and combine information from multiple sources, such as an insurance brokering system, an insurance liability computation, an automated travel planner, or a package tracking system. [26]

1.1.1 History of Web services

The Internet began its success story in the early nineties, even though it was used in the academic world before for many years. The main driver for the Internet's success was the World Wide Web, whose main innovation was the easy access to information, from any place, using standard Internet protocols and a simple data access protocol that enabled the implementation browsers on a variety of platforms. Together with the spread of the WWW, the Internet and its related technologies became the *de facto* standard to connect computers all around the world.

With the spread of the Internet, it became clear that the infrastructure that was introduced by the Internet could be used not just to retrieve information that was to be presented using a browser (called human-to-application, H2A, scenarios). Rather, there was also an increased demand for application-to-application (A2A) communication using the existing technologies. And, it was hoped that the existing protocols could be used for this purpose.

However, it soon became clear that this was not the case. HTTP had been designed with the retrieval of information in mind, following a very simple access path that basically relies on documents being linked together by means of hypertexts. The protocol does not cater for complex operations that arise from A2A scenarios. And some of the protocols that were defined at this time could not be used either because they did not fit into the Web World or they were too restrictive.

In late 1999 [32], Microsoft published an XML-based protocol, called SOAP that could be used for A2A scenarios. As it was one among many protocols suggested, it may be the fact that IBM started supporting SOAP in early 2000 that eventually lead to a public acceptance of SOAP by the industry.

At this point in time, SOAP was just a protocol to perform complex A2A scenarios. However, it quickly gained popularity and it was clear that there was a need for better describing and finding the services that were implemented using SOAP. The term Web services was coined several months later, when IBM, Microsoft, and Ariba jointly published the Web Services Description Language (WSDL). Eventually, UDDI (a Web service repository) was also introduced, thus completing the set of standards and protocols that make up the basis of Web services.

In the following years, many propositions were made about how to improve this technology such that it can be used not only for simple service invocation but also in more demanding environments. Among the most important ones, the Web services security (WSS) suite of standards is of particular interest, because it allows for a quality of service that is required by many enterprises and organizations. Since now, more than 40 specifications and standards have been published.

1.1.2 Types of Web services

In their simplest form [6], services are stateless, i.e. they provide a functional abstraction: the same service provides identical results, if it is invoked twice with the same arguments. If several instances of the same service are active for different clients, they all provide the same functional abstraction. An example can be a service that converts temperatures values from Celsius degrees to Fahrenheit. More complex services require some notion of state. It can be useful, however, to distinguish among different notions of stateful services.

Conversational services are yet another kind of stateful services. In a conversational service an instance of the service keeps a local state that depends on the conversation with its client. Every instance has its own local state, not visible to the others. Such kinds of services correspond to the well-known notion of data abstractions. Using familiar terms of object-oriented programming, each service can be viewed as an abstract data type and each instance as an object that can be

manipulated only through operations exposed in the service interface, which may modify the local state of the object. As an example of a conversational service, we consider the well-known shopping cart, available on most web sites providing e-commerce facilities. Each client has its own instance of the shopping cart, which contains the items selected by the client.

1.1.3 Differences between Web services and Web pages

Although Web pages provide access to applications across the Internet and across organizational boundaries, they are very different from Web services. Web pages are targeted at human users, whereas Web services are developed for access by humans as well as automated applications. As terminology is often used very loosely, it is easy to confuse someone by describing a “service” as a Web service when it is in fact not. Consequently, it is useful to examine first the concept of software-as-a-service on which Web services technology builds and then compare Web services to Web server-based functionality.

When comparing Web Services to Web-based applications we may distinguish four key differences:

- Web services act as resources to their applications that can request and initiate those Web services, with or without human intervention. This means that Web services can call on other Web services, providing a high degree of flexibility and adaptability not available in today’s Web-based applications.
- Web services are modular, self-aware, and self-describing applications; a Web service knows what functions it can perform and what inputs it requires to produce its outputs, and can describe this to potential users and to other Web services. A Web service can also describe its non-functional properties: for instance, the cost of invoking the service, the geographical areas the Web service covers, security measures involved in using the Web service, performance characteristics, contact information, and more.
- Web services are more visible and manageable than Web-based applications; the state of a Web service can be monitored and managed at any time by using external application management and workflow systems. Despite the

fact that a Web service may not run on a local system or may be written in an unfamiliar programming language, it still can be used by local applications, which may detect its state and manage the status of its outcome.

- Web services may be brokered or auctioned. If several Web services perform the same task, then several applications may place bids for the opportunity to use the requested service. A broker can base its choice on the attributes of the “competing” Web services, such as cost, speed and degree of security.

1.2 The Web services technology stack

The goal of Web services technology is to allow applications to work together over standard Internet protocols, without direct human intervention. By doing so, many business operations can be automated, creating new functional efficiencies and new, more effective ways of doing business. The minimum infrastructure required by the Web services paradigm is purposefully low to help ensure that Web services can be implemented on and accessed from any platform using any technology and programming language.

By intent, Web services are not implemented in a monolithic manner, but rather represent a collection of several related technologies. The more generally accepted definition for Web services leans on a stack of specific, complementary standards, which are illustrated in Figure 1.1. The development of open and accepted standards is a key strength of the coalitions that have been developing the Web services infrastructure; at the same time, as can be seen in Figure 1.1, these efforts have resulted in the proliferation of a dizzying number of emerging standards and acronyms. Below we provide a classification scheme for the most important standards in the Web services technology stack.

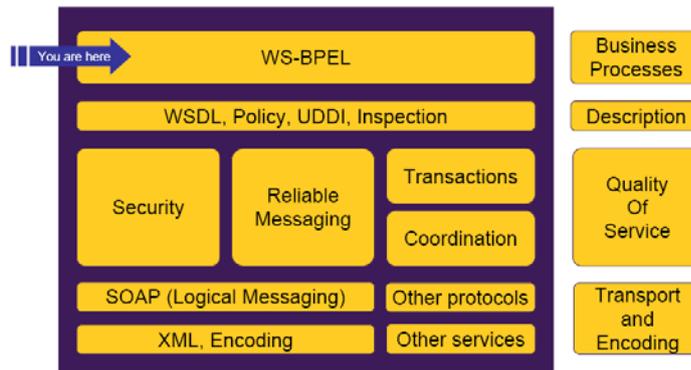


Figure 1.1: The Web Service technology stack

Enabling technology standards. Although not specifically tied to any specific transport protocol, Web services build on ubiquitous Internet connectivity and infrastructure to ensure nearly universal reach and support. For instance, at the transport level Web services take advantage of HTTP, the same connection protocol used by Web servers and browsers. Another enabling technology is the Extensible Markup Language (XML). XML is a widely accepted format for all exchanging data and its corresponding semantics. Web services use XML as the fundamental building block for nearly every other layer in the Web services stack.

Core services standards. The core Web services standards comprise the baseline standards SOAP, WSDL, and UDDI:

- *Simple Object Access Protocol:* SOAP is a simple XML-based messaging protocol on which Web services rely to exchange information among themselves. It is based on XML and uses common Internet transport protocols like HTTP to carry its data. SOAP implements a request/response model for communication between interacting Web services, and uses HTTP to penetrate firewalls, which are usually configured to accept HTTP and FTP service requests.
- *Service description:* Web services can be used effectively when a Web service and its clients rely on standard ways to specify data and operations, to represent contracts and to understand the capabilities a Web service provides. To achieve this, the functional characteristics of a Web service are first described by means of the Web Services Description Language. WSDL defines the XML

grammar for describing services as collections of communicating endpoints capable of exchanging messages.

- *Service publication:* Web service publication is achieved by UDDI, which is a public directory that provides publication of online services and facilitates eventual discovery of Web services. Companies can publish WSDL specifications for services they provide and other enterprises can access those services using the description in WSDL. In this way, independent applications can advertise their present business protocols or tasks that can be utilized by other remote applications and systems. Links to WSDL specifications are usually offered in an enterprise's profile in the UDDI registry.

Service composition and collaboration standards. These include the following standards:

- *Service composition:* Describes the execution logic of Web-services-based applications by defining their control flows (such as conditional, sequential, parallel, and exceptional execution) and prescribing the rules for consistently managing their unobservable business data. In this way enterprises can describe complex processes that span multiple organizations – such as order processing, lead management, and claims handling – and execute the same business processes in systems from other vendors. The Business Process Execution Language (BPEL) is the de-facto standard and is used to achieve service composition for Web services.
- *Service collaboration:* Describes cross-enterprise collaborations of Web service participants by defining their common observable behavior, where synchronized information exchanges occur through their shared contact points, when commonly defined ordering rules are satisfied. Service collaboration is materialized by the Web Services Choreography Description Language (WS-CDL), which specifies the common observable behavior of all participants engaged in business collaboration. Each participant could be implemented not only by BPEL but also by other executable business process languages.
- *Coordination/transaction standards:* Solving the problems associated with service discovery and service description retrieval is the key to success of Web services. Currently there are attempts underway towards defining transactional

interaction among Web services. The WS-Coordination and WS-Transaction initiatives complement BPEL to provide mechanisms for defining specific standard protocols for use by transaction processing systems, workflow systems, or other applications that wish to coordinate multiple Web services. These three specifications work in tandem to address the business workflow issues implicated in connecting and executing a number of Web services that may run on disparate platforms across organizations involved in e-business scenarios.

- *Value-added standards:* Additional elements that support complex business interactions must still be implemented before Web services can automate truly critical business processes. Value-added services standards include mechanisms for security and authentication, authorization, trust, privacy, secure conversations, contract management, and so on.

Today there are several vendors including companies such as IBM, Microsoft, BEA, and Sun Microsystems which supply products and services across the realm of Web services functionality and implement Web services technology stack. These vendors are considered as platform providers and provide both infrastructure, e.g. WebSphere, .NET framework, WebLogic, for building and deploying Web services in the form of application servers, as well as tools for orchestration and/or composite application development for utilizing Web services within business operations.

1.3 The need for monitoring

Once services and business processes become operational, their progress needs to be managed and monitored to gain a clear view of how services perform within their operational environment, take management decisions, and perform control actions to modify and adjust the behavior of Web-services-enabled applications. Service-level monitoring is a disciplined methodology for establishing acceptable levels of service that address business objectives, processes, and costs.

The service monitoring phase concerns itself with service-level measurement; monitoring is the continuous and closed loop procedure of measuring, monitoring, reporting, and improving the QoS of systems and applications delivered by service-oriented solutions. Service monitoring involves several distinct activities including logging and analysis of service execution details, obtaining business metrics by analyzing service execution data, detecting business situations that require management attention, determining appropriate control actions to take in response to business situations, whether in mitigating a risk or taking an opportunity, and using historical service performance data for continuous service improvement.

The service monitoring phase targets continuous evaluation of service-level objectives, monitoring the services layer for availability and performance, managing security policies, tracking interconnectivity of loosely coupled components, analyzing the root cause and correcting problems. To achieve this objective, service monitoring requires that a set of QoS metrics is gathered on the basis of SLAs, given that an SLA is an understanding of service expectations. In addition, workloads need to be monitored by the service provider to ensure that the promised performance level is being delivered, and to take appropriate actions to rectify non-compliance with an SLA, such as reprioritizing and reallocating resources.

To determine whether an objective has been met [26], SLA-available QoS metrics are evaluated based on measurable data about a service (e.g. response time, throughput, availability, and so on), performance during specified times, and periodic evaluations. SLAs include other observable objectives, which are useful for service monitoring. These include compliance with differentiated service-level offerings, i.e. providing differentiated QoS for various types of customers, individualized service-level offerings and requests policing which ensures that the number requests per customer stays within a predefined limit. All these also need to be monitored and assessed. A key aspect of defining measurable objectives is to set warning thresholds and alerts for compliance failures. For instance, if the response time of a particular service is degrading then the client could be automatically routed to a back up service.

1.4 Quality of service (QoS)

QoS refers to the ability of the Web service to respond to expected invocations and to perform them at the level commensurate with the mutual expectations of both its provider and its customers. Several quality factors that reflect customer expectations, such as constant service availability, connectivity, and high responsiveness, become key to keeping a business competitive and viable as they can have a serious impact upon service provision. QoS thus becomes an important criterion that determines the service usability and utility, both of which influence the popularity of a particular Web service, and an important selling and differentiating point between Web services providers.

Delivering QoS on the Internet is critical and significant because of its dynamic and unpredictable nature. Applications with very different characteristics and requirements compete for all kinds of network resources. Changes in traffic patterns, securing mission-critical business transactions, and the effects of infrastructure failures, low performance of Web protocols, and reliability issues over the Web create a need for Internet QoS standards. Often, unresolved QoS issues cause critical transactional applications to suffer from unacceptable levels of performance degradation.

Traditionally, QoS is measured by the degree to which applications, systems, networks, and all other elements of the IT infrastructure support availability of services at a required level of performance under all access and load conditions. While traditional QoS metrics can be applied, the characteristics of Web services environments bring both greater availability of applications and increased complexity in terms of accessing and managing services and thus impose specific and intense demands on organizations, which QoS must address. In the Web services' context, QoS can be viewed as providing assurance on a set of quantitative characteristics. These can be defined on the basis of important functional and non-functional service quality properties that include implementation and deployment issues as well as other important service characteristics such as service metering and cost, performance metrics (e.g. response time), security requirements, integrity, reliability, scalability, and availability. These characteristics are necessary requirements to understand the

overall behavior of a service so that other applications and services can bind to it and execute it as part of a business process.

The key elements for supporting QoS in a Web services environment are summarized in what follows [26]:

1. *Availability*: Availability is the absence of service downtimes. Availability represents the probability that a service is available. Larger values mean that the service is always ready to use while smaller values indicate unpredictability over whether the service will be available at a particular time. Also associated with availability is time-to-repair (TTR). TTR represents the time it takes to repair a service that has failed. Ideally smaller values of TTR are desirable.
2. *Accessibility*: Accessibility represents the degree with which a Web service request is served. It may be expressed as probability measure denoting the success rate or chance of a successful service instantiation at a point in time. A high degree of accessibility means that a service is available for a large number of clients and that clients can use the service relatively easily.
3. *Conformance to standards*. Describes the compliance of a Web service with standards. Strict adherence to correct versions of standards by service providers is necessary for proper invocation of Web services by service requestors. In addition, service providers must stick to the standards outlined in Service-Level Agreements between service requestors and providers.
4. *Integrity*. Describes the degree with which a Web service performs its tasks according to its WSDL description as well as conformance with Service-Level Agreement (SLA). A higher degree of integrity means that the functionality of a service is closer to its WSDL description or SLA.
5. *Performance*. Performance is measured in terms of two factors: throughput and latency. *Throughput* represents the number of Web service requests at a given time period. *Latency* represents the length of time between sending a request and receiving the response. Higher throughput and lower latency values represent good performance of a Web service. When measuring the

transaction/request volumes handled by a Web service it is important to consider whether these come in a steady flow or burst around particular events like the open or close of the business day or seasonal rushes.

6. *Reliability*. Reliability represents the ability of a service to function correctly and consistently and provide the same service quality despite system or network failures. The reliability of a Web service is usually expressed in terms of number of transactional failures per month or year.
7. *Scalability*. Scalability refers to the ability to consistently serve the requests despite variations in the volume of requests. High accessibility of Web services can be achieved by building highly scalable systems.
8. *Security*. Security involves aspects such as authentication, authorization, message integrity, and confidentiality. Security has added importance because Web service invocation occurs over the Internet. The amount of security that a particular Web service requires is described in its accompanying SLA, and service providers must maintain this level of security.
9. *Transactional*. There are several cases where Web services require transactional behavior and context propagation. The fact that a particular Web service requires transactional behavior is described in its accompanying SLA, and service providers must maintain this property.

1.5 Service-Level Agreements (SLAs)

As organizations depend on business units, partners, and external service providers to furnish them with services, they rely on the use of SLAs to ensure that the chosen service provider delivers a guaranteed level of service quality. An SLA sets the expectations between the consumer and provider. It helps define the relationship between the two parties. It is the cornerstone of how the service provider sets and maintains commitments to the service consumer. A properly specified SLA describes each service offered and addresses:

- how delivery of the service at the specified level of quality will become realized

- which metrics will be collected
- who will collect the metrics and how
- actions to be taken when the service is not delivered at the specified level of quality and who is responsible for doing them
- penalties for failure to deliver the service at the specified level of quality
- how and whether the SLA will evolve as technology changes (e.g., multi-core processors improve the provider's ability to reduce end-to-end latency) [3]

In the definition of an SLA, realistic and measurable commitments are important. Performing as promised is important, but swift and well communicated resolution of issues is even more important.

The challenge [33] for a new service and its associated SLA is that there is a direct relationship between the architecture and the maximum levels of availability. Thus, an SLA cannot be created in a vacuum. An SLA must be defined with the infrastructure in mind. An exponential relationship exists between the levels of availability and the related cost. Some customers need higher levels of availability and are willing to pay more. Therefore, having different SLAs with different associated costs is a common approach.

An SLA may contain the following parts:

- *Purpose*: This field describes the reasons behind the creation of the SLA.
- *Parties*: This field describes the parties involved in the SLA and their respective roles, e.g. service provider and service consumer (client).
- *Validity period*: This field defines the period of time that the SLA will cover. This is delimited by start and end time of the agreement term.
- *Scope*: This field defines the services covered in the agreement.
- *Restrictions*: This field defines the necessary steps to be taken in order for the requested service levels to be provided.
- *Service-level objectives*: This field defines the levels of service that both the service customers and the service providers agree on, and usually includes a set of service level indicators, like availability, performance, and reliability. Each of these aspects of the service level will have a target level to achieve.

- *Penalties*: This field defines what sanctions should apply in case the service provider underperforms and is unable to meet the objectives specified in the SLA.
- *Optional services*: This field specifies any services that are not normally required by the user, but might be required in case of an exception.
- *Exclusion terms*: These specify what is not covered in the SLA.
- *Administration*: This field describes the processes and the measurable objectives in an SLA and defines the organizational authority for overseeing them.

SLAs can be either static or dynamic in nature. A static SLA is an SLA that generally remains unchanged for multiple service time intervals. Service time intervals may be calendar months for a business process that is subject to an SLA, or may be a transaction or any other measurable and relevant period of time for other processes. They are used for assessment of the QoS and are agreed between a service provider and service client. A dynamic SLA is an SLA that generally changes from service period to service period, to accommodate changes in provision of service.

To enter into a Web services SLA, specific QoS metrics that are evaluated over a time interval to a set of defined objectives should be employed. Measurement of QoS levels in an SLA will ultimately involve tracing Web services through multi-domain (geographical, technological, application, and supplier) infrastructures. In a typical scenario, each Web service may interact with multiple Web services, switching between the roles of being a service provider in some interactions to being a consumer in other interactions. Each of these interactions could potentially be governed by an SLA. The metrics imposed by SLA should correlate with the overall objectives of the services being provided. Thus an important function that an SLA accomplishes is addressing QoS at the source. This refers to the level of service that a particular service provides. [26]

The credibility [16] of SLAs is essential for the functioning of such a service market. Unreliable SLA advertisements decrease the overall welfare of the market, since clients do not have accurate information to plan their business. As clients are usually required to pay for an SLA before receiving the requested service, providers have an opportunity to cheat. They may provide lower QoS than advertised, and thus

save costs. It is therefore necessary to create incentives for service providers to respect their advertised SLAs by stating penalties that must be paid when the delivered QoS is less than promised.

1.5.1 Why a Service Level Agreement is Important

A good SLA is important [33] because it sets boundaries and expectations for the following aspects of data center service provisioning.

- *Customer commitments.* Clearly defined promises reduce the chances of disappointing a customer. These promises also help to stay focused on customer requirements and assure that the internal processes follow the right direction.
- *Key performance indicators for the customer service.* By having these indicators established, it is easy to understand how they can be integrated in a quality improvement process. By doing so, improved customer satisfaction stays a clear objective.
- *Key performance indicators for the internal organizations.* An SLA drives internal processes by setting a clear, measurable standard of performance. Consequently, internal objectives become clearer and easier to measure.
- *The price of non-conformance.* If the SLA has penalties non-performance can be costly. However, by having penalties defined, the customer understands that the provider truly believes in its ability to achieve the set performance levels. It makes the relationship clear and positive.

1.5.2 SLA Life Cycle

Service Level Agreements have a certain life cycle, which they are running through from the initial preparation of the templates for an agreement up to the evaluation whether the agreement has been fulfilled, or partly or completely violated. W. Sun et al [5] describe that the SLA life cycle consists of six phases; these phases are shown in Figure 1.2.

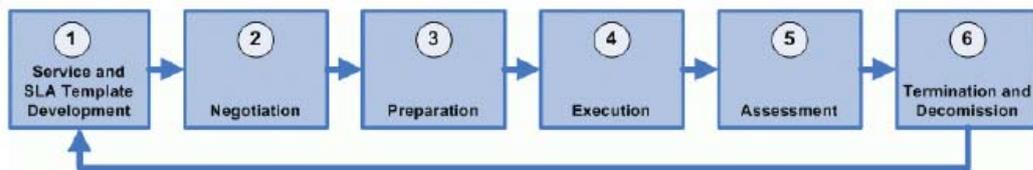


Figure 1.2: SLA Life Cycle

- 1) *Service and SLA Template Development*: This phase includes the identification of service consumer needs, the identification of appropriate service characteristics and parameters that can be offered given the service execution environment, and the preparation of standard SLA templates.
- 2) *Negotiation*: This phase includes the negotiation of the specific values for the defined service parameters, the costs for the service consumer, the costs for the service provider when the SLA is violated, and the definition and periodicity of reports to be provided to the service consumer.
- 3) *Preparation*: The service (or a specific instance of it) is prepared for consumption by the service consumer. This phase may require the reconfiguration of the resources that support service execution in order to meet SLA parameters.
- 4) *Execution*: This phase is the actual operation of the service. It includes service execution and monitoring, real-time reporting, service quality validation, and real-time SLA violation processing.
- 5) *Assessment*: This phase has two parts:
 - a) Assessment of the SLA and the QoS that is provided to an individual consumer. QoS, consumer satisfaction, potential improvements, and changing requirements are reviewed periodically for each SLA.
 - b) Assessment of the overall service. This assessment can be tied to an internal business review. Elements to be covered in this review are the QoS provided to all consumers, the need for the realignment of service goals and operations,

the identification of service support problems, and the identification of the need for different service levels.

- 6) *Termination and Decommission*: This phase deals with termination of the service for reasons such as contract expiration or violation of contract, as well as the decommission of discontinued services.

1.5.3 Qualities That Can Be Defined in an SLA

In theory, it is possible to specify any quality in an SLA [8], provided that all parties understand how to measure or verify its achievement. We've seen two categories of qualities that can be specified in SLAs: measurable and unmeasurable. Measurable qualities can be measured automatically using metrics; for example, the percentage of time a system is available. Unmeasurable qualities are those that cannot be measured automatically from a given viewpoint; for example, determining the cost of changing a service is difficult to automate.

Measurable Qualities

- *Accuracy* is concerned with the error rate of the service. It is possible to specify the average number of errors over a given time period.
- *Availability* is concerned with the mean time to failure for services, and the SLAs typically describe the consequences associated with these failures. Availability is typically measured by the probability that the system will be operational when needed. It is possible to specify
 - the system's response when a failure occurs
 - the time it takes to recognize a malfunction
 - how long it takes to recover from a failure
 - whether error handling is used to mask failures
 - the downtime necessary to implement upgrades (may be zero)
 - the percentage of time the system is available outside of planned maintenance time

- *Capacity* is the number of concurrent requests that can be handled by the service in a given time period. It is possible to specify the maximum number of concurrent requests that can be handled by a service in a set block of time.
- *Cost* is concerned with the cost of each service request. It is possible to specify
 - the cost per request
 - the cost based on the size of the data
 - cost differences related to peak usage times
- *Latency* is concerned with the maximum amount of time between the arrival of a request and the completion of that request.
- *Provisioning-related time* (e.g., the time it takes for a new client's account to become operational)
- *Reliable messaging* is concerned with the guarantee of message delivery. It is possible to specify
 - how message delivery is guaranteed (e.g., exactly once, at most once)
 - whether the service supports delivering messages in the proper order
- *Scalability* is concerned with the ability of the service to increase the number of successful operations completed over a given time period. It is possible to specify the maximum number of such operations.

Unmeasurable Qualities

- *Interoperability* is concerned with the ability of a collection of communicating entities to share specific information and operate on it according to an agreed-upon operational semantics. It is possible to specify the standards supported by the service and to verify them at runtime. Significant challenges still need to be overcome to achieve semantic interoperability at runtime.
- *Modifiability*, in this context, is concerned with how often a service is likely to change. It is possible to specify how often the service's
 - interface changes
 - implementation changes
- *Security* is concerned with the system's ability to resist unauthorized usage, while providing legitimate users with access to the service. Security is also characterized as a system providing non-repudiation, confidentiality, integrity, assurance, and auditing. It is possible to specify the methods for

- authenticating services or users
- authorizing services or users
- encrypting the data

1.6 Contributions

In a nutshell, the main contributions of this thesis are:

- We propose a system that combines the web service monitoring with SLA compliance.
- We study the ways the metrics for composed web services can be calculated, having available only the metrics of the constituent services.
- We propose types for calculating the response time, the throughput, the availability and the cost of composed services.
- Interesting experimental results are reported.

1.7 Organization of the thesis

Chapter 2 introduces the fundamentals about monitoring. We discuss why the monitoring is appropriate in SOA systems, we mention which services are involved in the monitoring procedure and we analyze the use of WSLA for expressing SLAs. Finally, we briefly mention the implementation procedure of our work and the tools, used for it.

Chapter 3 presents the basic composition patterns. Furthermore, we propose some formulas for computing the response time, the throughput, the reliability and the cost of complex Web services. A use case in the end clarifies the afore-mentioned work.

Chapter 4 examines the state of the art approaches and tools used for monitoring Web services and computing metrics for composed services.

Chapter 5 presents an experimental analysis of our work. We use some simple experiments that validate our work.

Chapter 6 summarizes the results of this thesis and identifies topics that are worth further work and research.

Chapter 2

2 Monitoring of Web services

2.1 Introduction

Monitoring consists of a verification at run-time that the requirements, specified by the clients and by the service providers, are met during execution. The requirements can obviously be of very diverse nature. There are three complementary dimensions to the coexisting monitoring problem: [14]

- *Assertion-based monitoring*
- *Event-based monitoring*
- *History-based monitoring*

Assertion-based monitoring consists of asking the BPEL process to use an external monitor service to verify the correctness of certain assertions at given points of the process execution. The assertions that must be monitored are the pre- and post-conditions derived from the conjunction of those provided by the service provider and those provided by the client of the process. Starting from a complete BPEL process—the unmonitored process— assertions are added to the process in the form of commented annotations. The location in which they are added indicates where in a process an assertion must be checked. Once an annotated version of the unmonitored process is available, it is passed through a transformation algorithm called BPEL2BPEL, which produces what we call the monitored process. The algorithm adds BPEL code to the process to support the use of an external monitor service to check the validity of the assertions. The added code prepares the message to be sent to the external monitor, sends it, and verifies the monitor’s answer. The message that

is sent to the external monitor service contains two items: the assertion to be checked and the data on which the assertions must be checked. These data typically consist of information about the internal state of the process in execution. Depending on the implementation of the external monitor, the data on which to verify the assertions might also be obtained elsewhere.

This approach obviously has a substantial impact on performance, since the process execution is momentarily stopped to execute the monitoring. However, it is simply a technique that is offered to the process designer, which can be used as needed. For example, it might be used only sporadically or even be switched off totally or partially at any time. Its strength is that, since the monitoring is inserted in the form of annotations, the business logic remains separate from the monitoring logic up until when the BPEL2BPEL performs the transformation that weaves the two to produce a new version of the monitored process. The original version is not touched. Second, the resulting process remains pure BPEL code and is executable on any standard BPEL engine. The inter-weaving should be able to take into account the different monitoring necessities of different stake-holders and/or different times of the life-cycle of the process. It must be noted that in this approach it is easier to monitor functional contracts, with respect to non-functional contracts.

Event-based monitoring comes into play when it is necessary to verify non-functional qualities of a service or of a composition. It consists of a less-intrusive approach to monitoring where, in parallel to the execution of a business process, a monitoring component can listen to the events launched by the BPEL engine. Since this is done in parallel to the execution, almost no impact on performance is expected. Obviously, this approach is closely tied to the particular implementation of the BPEL engine in use. For example, using ActiveBPEL, an open source BPEL engine, it is possible to listen to a series of events that the engine provides. In particular, these events are tied to the standard BPEL activities (i.e. invoke, receive, etc.) and to their state changes. Each activity can be, for example, in a Ready to execute state, in an Executing state, in a Terminated state or in a Faulted state.

Another possible approach would be to position an observer service between the BPEL engine and the outside world and to monitor the contents of the SOAP messages flowing in and out of the system. Both approaches must be further

investigated in order to discover the strengths and weaknesses of each. At the moment, it is clear that both work at a lower-level with respect to the BPEL process and to the assertion-based approach in monitoring. As a consequence, once an erroneous behavior is observed, it is not possible to simply freeze the execution of the process to implement a recovery action, like dynamic reconfiguration of the process. Intervention in the execution could be obtained by selecting points in the process in which to introduce an assertion. This assertion could represent a synchronization point between the process execution and the collection of events. By adding an assertion, we would be effectively mixing event-based monitoring with assertion-based monitoring. So, if the assertion is false, at least one of the checked LTL formulas must be false, meaning recovery actions should be taken.

History-based monitoring is an extension to event-based monitoring. By collecting events in a history event repository it is possible to reason upon QoS requirements that deal with a history of process executions. An example of such a requirement could be “eighty percent of the times a process goes into execution it must complete within one minute”. The event-based approach and the history-based approach still have to be studied in depth, although it already seems clear that the three should coexist to obtain complete functional and/or non-functional monitoring.

2.2 Services involved in compliance monitoring

During the contracting process, after the main elements of the SLA are agreed upon, customer and provider may define third parties (in the WSLA context we refer to these as *supporting parties*) to which SLA monitoring tasks may be delegated.

When the SLA is finalized, both provider and customer make the SLA document available for deployment. The *deployment service* is responsible for checking the validity of the SLA and distributing it either in full or in part to the supporting parties.

Figure 2.1 [8] illustrates the services involved in compliance monitoring when multiple parties are involved. Services that may be outsourced to third parties are either measurement services or condition evaluation services.

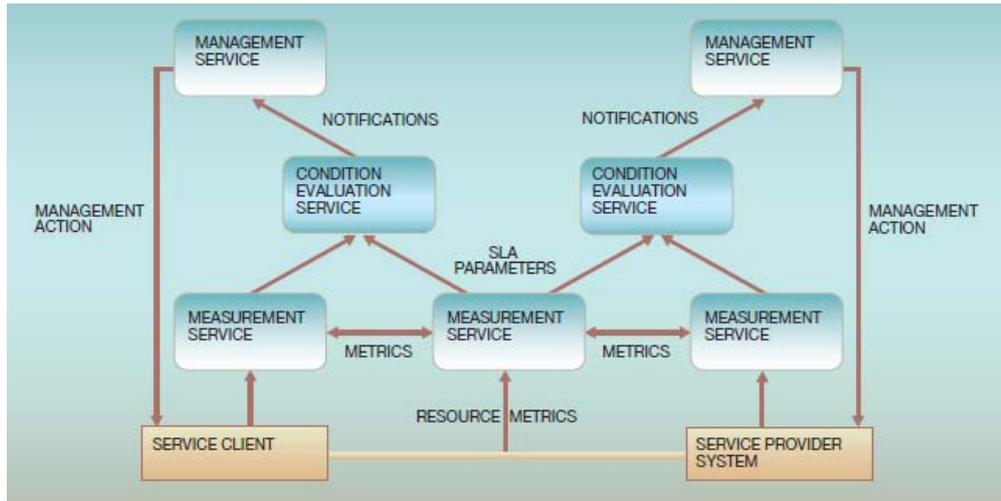


Figure 2.1: Services involved in SLA-compliance monitoring with multiple parties

The *measurement* service maintains information on the current system configuration and runtime information on the metrics that are part of the SLA. It measures SLA parameters, such as availability or response time, either from inside, by retrieving resource metrics directly from managed resources, or from outside the service provider’s domain, for example, by probing or intercepting client transactions. A measurement service may measure all or a subset of the SLA parameters. Multiple measurement services may simultaneously measure the same metrics, e.g., a measurement service may be located within the provider’s domain while another measurement service probes the service offered by the provider across the Internet from various locations.

As depicted in Figure 2.1, measurement services may be cascaded, that is, a third measurement service may be used to aggregate data computed by other measurement services. In this way, we refer to metrics that are retrieved directly from managed resources as *resource metrics*. *Composite metrics*, in contrast, are created by aggregating several resource (or other composite) metrics according to a specific algorithm, such as averaging one or more metrics over a specific amount of time or by breaking them down according to specific criteria (e.g., top 5 percent, minimum, maximum, mean, median etc.). This is usually done by a measurement service within a service provider’s domain, but can be outsourced to a third-party measurement service as well.

The *condition evaluation service* is responsible for monitoring compliance of the SLA parameters at runtime with the agreed-upon service level objective (SLO)

by comparing measured parameters against the thresholds defined in the SLA and notifying the management services of the customer and the provider. It obtains measured values of SLA parameters from one or more measurement services and tests them against the guarantees given in the SLA. This can be done each time a new value is available, or periodically.

Finally, both service customer and provider have a *management service*. Upon receipt of a notification, the management service takes appropriate actions to correct a problem, as specified in the SLA. The main purpose of the management service is to execute corrective actions on behalf of the managed environment if a condition evaluation service discovers that a term of an SLA has been violated.

2.3 Monitoring and Managing SLAs

Figure 2.2 shows a conceptual architecture for an SLA monitoring and management infrastructure. Instrumentation is added to the service user and to the service provider to generate metrics. The measurement subsystem receives the metrics from instrumentation on one or more components and computes or combines data to produce values for the parameters specified in the SLA (i.e., the service level parameters). Figure 2.2 also shows measurement subsystem components on the service user and service provider sides, but there are also other alternatives. The measurement subsystem could exist only on the service user side or only on the service provider side, but the party providing measurement needs to be trusted by the other party. Measurement could also be partially or totally implemented by a third-party component running on a separate machine. As Figure 2.2 illustrates, the values of the SLA parameters are input for the evaluation procedure, which can run on

- either the service user or service provider
- both the service user and service provider
- a third-party machine



Figure 2.2: Conceptual Architecture for SLA Monitoring and Management

The evaluation procedure checks the values against the guaranteed conditions of the SLA. If any value violates a condition in the SLA, predefined actions are invoked. Action-handler components are responsible for processing each action request. While the functionality they execute will vary, it will likely include some form of violation notification or recording. Action handlers may exist on the service user machine, service provider machine, and/or third-party machines.

An important output of SLA management not shown in Figure 2.2 consists of reports that contain analyses of the metrics and thresholds. These reports can guide planning and implementation of improvements to the service infrastructure.

The same service may be invoked by different service users under different service level agreements. In that case, each service call may need to identify the agreement that applies to that invocation. Alternatives to agreement identification include tagging service calls with agreement identifiers, identifying the service user based on information exchanged at binding time, and selecting agreements based on the user.

Expressing SLAs in the WSLA Language

In this section, we provide a brief overview over the parts of the WSLA language that relate to the definition of SLA parameters and the way they are monitored.

The *Parties* [12] section identifies the contractual parties and contains the technical properties of a party, i.e. their address and interface definitions (e.g. the ports for receiving notifications). The *Service Description* section of the SLA specifies the characteristics of the service and its observable parameters as follows: For every *Service Operation*, one or more *Bindings*, i.e., the transport encoding for the messages to be exchanged, may be specified. In addition, one or more *SLA Parameters* of the service may be specified. Examples of such SLA parameters are *service availability, throughput, or response time*. Every SLA parameter refers to one *Metric*, which, in turn, may aggregate one or more other (composite or raw) metrics, according to a measurement directive or a function. Examples of composite metrics are *maximum response time of a service, average availability of a service, or minimum throughput of a service*. Examples of raw metrics are: *system uptime, service outage period, number of service invocations*. *Measurement Directives* elements are used when the value of a Metric should be measured directly from a resource by probing or instrumentation of the system. There are seven types of measurement directives in the WSLA specification. Typical examples of measurement directives [17] are the uniform resource identifier of a hosted computer program, a protocol message (e.g., an SNMP GET message), the command for invoking scripts or compiled programs, or a query statement issued against a database or data warehouse. *Function* elements are used for a metric if the metric value is derived from the value of other metrics or constants. There are eighteen types of functions used in WSLA document. Examples of functions are formulas of arbitrary length containing mean, median, sum, minimum, maximum, and various other arithmetic operators, or time series constructors. For every function, a *Schedule* is specified. It defines the time intervals during which the functions are executed to retrieve and compute the metrics. These time intervals are specified by means of *start time, duration, and frequency*. Examples of the latter are *weekly, daily, hourly, or every minute*. *Obligations*, the last section of an SLA, define the SLOs, guarantees and constraints that may be imposed on the SLA parameters.

2.3.1 Why we use WSLA

In fact, there are many approaches towards the QoS specification and management for Web services. In this subsection we provide more details for five of them and we conclude to our choice to choose WSLA for our work. These five approaches are:

- the Web Service Level Agreement (WSLA) developed by IBM,
- the Web Service Offering Language (WSOL) developed at Carleton University of Canada
- SLang developed at University College London, UK
- a UDDI eXtension (UX) developed at Nanyang Technological University, Singapore, and
- UDDIe developed at Cardiff University, UK.

Web Service Level Agreement (WSLA)

As already described, WSLA was developed by IBM and is used to define SLA documents [18]. A WSLA is an agreement between a service provider and a customer and as such defines the obligations of the parties involved. Primarily, it is the obligation of a service provider to perform a service according to agreed-upon guarantees for the service parameters on the technical level.

The design goals of WSLA are a formal and flexible XML-based language for SLA definitions between different organizations, a wide acceptance and applicability to existing e-business systems and standards, nested relationships of service clients and providers, delegation of monitoring tasks to third parties, and an SLA-driven configuration of the managed resources, i.e. deriving configuration settings directly from SLAs.

Web service offering language (WSOL)

A research group from Carleton University in Canada has developed [31] the notion of providing various *classes of service* for one and the same functional service specification, which differ in QoS level and management efforts. WSOL allows the formal and unambiguous specification of prices, monetary penalties, management responsibilities and third parties, especially accounting parties.

The main targets of the WSOL project are the creation of service offerings, definition of QoS constraints, management statements, reusability, and a mechanism called service offering dynamic relationship (SODR) allowing for switching between services.

Another important design goal is a low run-time overhead achieved through defining classes of services instead of individually managed SLAs. WSOL also supports reusability of specifications. This is realized by means of the concept of constraint groups and constraint group templates to include formerly defined elements and import of elements defined in other WSOL files.

SLAng

As an XML-based language for defining service level agreements, SLAng [20], was developed at the University College London, UK. The main targets of SLAng are middleware, and applications as well as the specification of non-functional parameters at service level in order to enable QoS description and negotiation.

At the moment SLAng can be used only for static SLAs, since it does not support dynamic lookup of new services and update of non-functional service properties at runtime.

UX

UX [34] is an architecture providing QoS-aware and cross organizational support for UDDI, developed at the School of Computer Engineering, Nanyang Technological University, Singapore. The first goal of UX is to rate services with

reputation in order to allow service requestors to discover services with good quality. The second one is to share the ratings among UX servers in different domains.

UDDIe

UDDIe [30] was developed at Cardiff University, UK. It extends UDDI's functionalities within UDDI. Service providers can associate their services with QoS properties such as bandwidth, CPU, and memory requirements, which are encoded in the service interface. They can make their services available for a period of time by means of leasing. UDDIe introduced a concept allowing the definition of three leasing types including finite, infinite, and future lease. Furthermore, UDDIe supports qualifier-based search by introducing qualifiers such as *EQUALto*, *LESSthan*, and *GREATERthan*.

Comparison of approaches

Here we make a comparison of the previous approaches, used for specifying and managing SLAs. The assessment criteria used are requirement specification, class of service, QoS aspects, QoS mapping, and flexibility:

Requirement specification: Both Web service clients and providers need to specify non-functional requirements and offers. The specification should ensure the compatibility and comparability of the specifications done by clients and service providers.

Class of service: QoS parameters differ in quality, quantity, and the corresponding monetary charge. Grouping similar parameters into a class or category that characterize a service will ease the utilization of the service.

QoS aspects: A Web services related framework should support more than the classical QoS parameters such as jitter and bandwidth. Aspects such as security, reliability, transaction as well as custom defined aspects should also be considered.

QoS mapping: An overall QoS support requires QoS support during the whole communication process, ranging from the QoS specification to monitoring at runtime. QoS has also to be considered through the different layers in terms of the

Internet Model. Specifications in higher layers have to be carefully mapped onto lower layers.

Flexibility: An approach should be easy to use, extensible, and standards conforming.

The assessment of the introduced approaches is summarized in Table 1. The symbols mean:

“++”: excellent concept

“+”: good concept

“O”: satisfying

“-”: poor or not available

	Requirements specification	Class of service	QoS aspects	QoS mapping	Flexibility
WSLA	++	O	+	-	++
WSOL	++	++	+	-	+
SLang	+	-	+	-	+
UX	O	-	O	-	O
UDDIe	O	-	O	-	O

Table 2.1: Assessment of the introduced approaches

From the table above we can conclude that none of the introduced approaches satisfy all the assessment criteria. WSLA and WSOL are the best solutions as they meet almost all criteria, except QoS mapping that is not supported. Slang is a “medium” solution as it has a good concept of three criterias and UX and UDDIe are the least used approaches, as they only have a satisfying concept of three criterias.

From the first two approaches, we used WSLA because it supports the definition of Service Level Objectives that are appropriate in our work. In addition, in WSLA all the metrics with the agreed values are defined explicitly, in contrast to WSOL that supports only the creation of service offerings and definition of QoS constraints. Finally, WSLA is an XML-based language that is very easy in use and is supported by many tools.

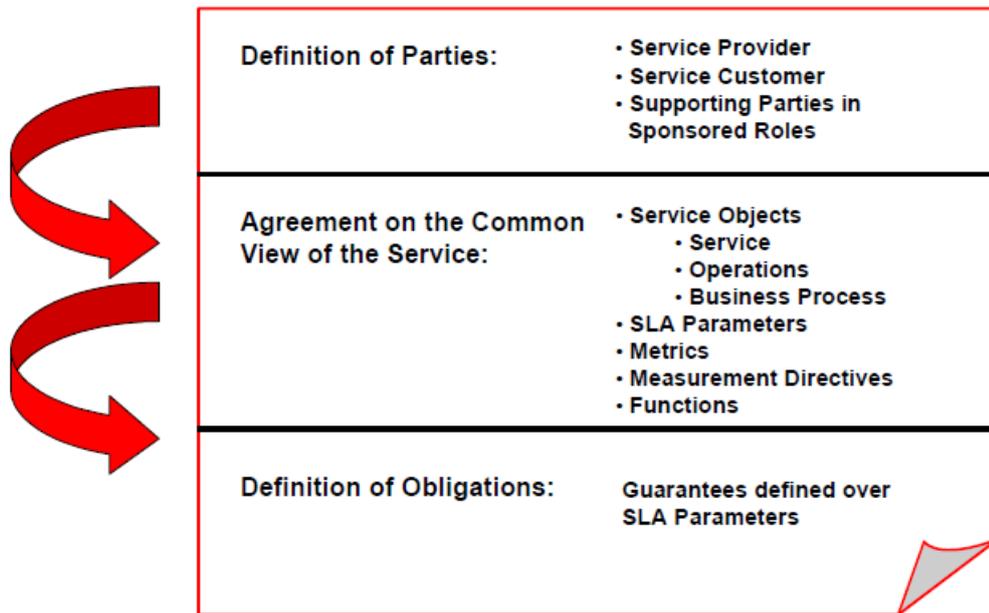


Figure 2.3: SLA structure as defined in WSLA

2.3.2 SLA Parameters and metrics

SLA parameters [18] are properties of a service object; each SLA parameter has a name, type and unit. SLA parameters are computed from metrics which either define how a value is to be computered from other metrics or describe how it is measured. For this purpose, a metric either defines a function that can use other metrics as operands or it has a measurement directive that describes how the metric's value should be measured. Since SLA parameters are the entities that are surfaced by a Measurement Directive to a Condition Evaluation Service, it is important to define which party is supposed to provide the value (source) and which parties can receive it, in an event-driven manner (push) or through polling (pull). In Figure 2.4, one metric is retrieved by probing an interface (service probe) while the other ones (TXcount, Timecount) are directly retrieved from the service provider's management system.

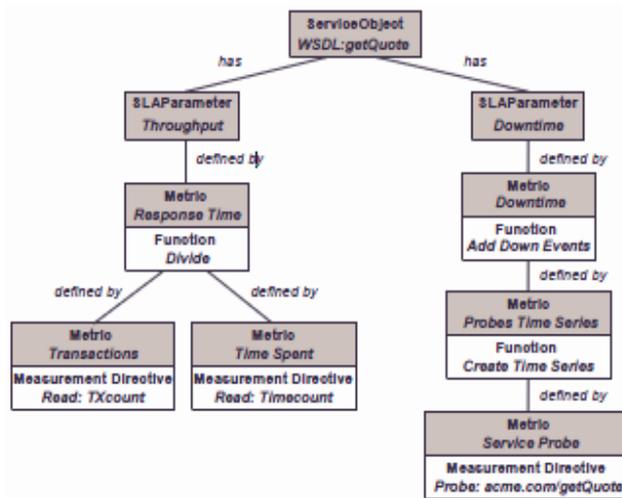


Figure 2.4: Sample Elements of Service Description

A metric's purpose is to define how to measure or compute a value. Besides a name, a type and a unit, it contains either a function or a measurement directive and a definition of the party that is in charge of computing this value. Figure 2.5 shows an example of composite metric, containing a function.

```

<Metric name=" AverageResponseTime"
  type="double" unit="seconds">
  <Source>ACMEProvider</Source>
  <Function xsi:type="Divide"
    resultType="double">
    <Operand>
      <Function xsi:type="TSSelect"
        resultType="double">
        <Operand>
          <Metric>TimeSpentTS</Metric>
        </Operand>
        <Element>0</Element>
      </Function>
    </Operand>
    <Operand>
      <Function xsi:type="TSSelect"
        resultType="double">
        <Operand>
          <Metric>TXCountTS</Metric>
        </Operand>
        <Element>0</Element>
      </Function>
    </Operand>
  </Function>
</Metric>

```

Figure 2.5: Metric example in WSLA

The example above describes the average response time metric. The metric is of type double and its unit is seconds. ACMEProvider will measure its value. In its function definition, the function Divide is applied to two operands, which in turn are again functions. The function TSSelect yields elements of a time series. The element

“0” means the most recent value. Specific functions, such as Minus, Plus or TSSelect are extensions of the common function type. Operands of functions can be metrics, scalars and other functions. It is expected that a measurement service implementation is able to compute functions. Specific functions can be added to the standard set as needed.

```
<SLAParameter name="AverageResponseTime"
               type="float"
               unit="seconds">
  <Metric>AverageResponseTime</Metric>
  <Communication>
    <Service>ACMEProvider</Service>
    <Pull>XYZAuditing</Pull>
    <Push>ACustomer</Push>
  </Communication>
</SLAParameter>
```

Figure 2.6: SLA parameter example in WSLA

2.3.3 Obligations

Based on the common terminology established in the service definition part of the WSLA document, the parties can unambiguously define the respective guarantees that they give to each other. The obligations section of the SLA may contain any number of guarantees. The WSLA language provides two kinds of guarantees:

- **Service level objectives** represent promises with respect to the state of SLA parameters.
- **Action guarantees** are promises to perform an action. This may include notifications of service level objective violations or invocation of management operations. Important for both types of guarantees is the definition of the obliged party and the definition of when they need to be evaluated. The actual definition of the guarantees' content is specific to each type.

2.3.4 Service level objectives

A service level objective expresses a commitment to maintain a particular state of the service in a given period. Any party can take the obliged part of this

guarantee. However, this is typically the service provider. A service level objective has the following elements:

- The **Obligated** is the name of a party that is in charge of delivering what is promised in this guarantee.
- One or many **ValidityPeriods** define when the guarantee is applicable.
- A logic **Expression** defines the actual content of the guarantee, i.e., what is asserted by the service provider to the service customer. A logic expression follows first order logic. Expressions contain the usual operators *and*, *or*, *not*, etc., which connect predicates or, again, expressions. **Predicates** can have SLA parameters and scalar values as parameters. By extending an abstract predicate type, new domain-specific predicates can be introduced as needed. Similarly, expressions could be extended e.g., to contain variables and quantifiers. This provides the parties the expressiveness to define complex states of the service.
- A service level objective may have an **EvaluationEvent**, which defines when the expression of the service level objective should be evaluated. The most common evaluation event is *NewValue*, each time a new value for an SLA parameter used in a predicate is available.
- Alternatively, the expression may be evaluated according to a **Schedule**. A schedule is a sequence of regularly occurring events. It can be defined within a guarantee or a commonly used schedule can be referred to.

The example of figure 2.7 illustrates service level objectives:

```

<ServiceLevelObjective name="slol">
  <Obligated>ACMEProvider</Obligated>
  <Validity>
    <Start>2002-11-30T14:00:00.000-05:00</Start>
    <End>2002-12-31T14:00:00.000-05:00</End>
  </Validity>
  <Expression>
    <Implies>
      <Expression>
        <Predicate xsi:type="Less">
          <SLAParameter>Transactions</SLAParameter>
          <Value>10000</Value>
        </Predicate>
      </Expression>
      <Expression>
        <Predicate xsi:type="Less">
          <SLAParameter>AverageResponseTime</SLAParameter>
          <Value>0.5</Value>
        </Predicate>
      </Expression>
    </Implies>
  </Expression>
  <EvaluationEvent>NewValue</EvaluationEvent>
</ServiceLevelObjective>

```

Figure 2.7: Service Level Objective example in WSLA

The example shows a service level objective given by ACMEProvider for one month in 2002. It guarantees that the SLA parameter AverageResponseTime must be less than 0.5 if the SLA parameter Transactions is less than 10000. This condition should be evaluated each time a new value for the SLA parameter is available.

2.3.5 Action guarantees

An action guarantee [9] expresses a commitment to perform a particular activity if a given precondition is met. Any party can be the obliged of this kind of guarantee. This particularly includes also the supporting parties of the contract. An action guarantee comprises of the following elements and attributes:

- The **Obligated** is the name of a party that must perform an action as defined in this guarantee.
- A logic **Expression** defines the precondition of the action. The format of this expression is the same as the format of expression in service level objectives. An important predicate for action guarantees is the Violation predicate that determines whether another guarantee, in particular a service level objective, has been violated.

- An **EvaluationEvent** or an evaluation **Schedule** defines when the precondition is evaluated.
- The **QualifiedAction** contains a definition of the action to be invoked at a particular party. The concept of a qualified action definition is similar to the invocation of an object method in a programming language, replacing the object name with a party name. The party of the qualified action can be the obliged or another party. The action must be defined in the corresponding party specification. In addition, the specification of the action includes the marshalling of its parameters. One or more qualified actions can be part of an action guarantee.
- The **ExecutionModality** is an additional means to control the execution of the action. It can be defined whether the action should be executed if a particular evaluation of the expression yields true. The purpose is to reduce, for example, the execution of a notification action to a necessary level if the associated expression is evaluated very frequently. Execution modality can be either: *always*, *on entering a condition* or *on entering and leaving a condition*.

The following example (figure 2.8) illustrates an action guarantee:

```

<ActionGuarantee name="ag2">
  <Obligated>XYZAuditing</Obligated>
  <Expression>
    <Predicate xsi:type="Violation">
      <ServiceLevelGuarantee>slol</ServiceLevelGuarantee>
    </Predicate>
  </Expression>
  <EvaluationEvent>NewValue</EvaluationEvent>
  <QualifiedAction>
    <Party>ACustomer</Party>
    <Action actionName="notification"
      xsi:type="Notification">
      <NotificationType>Violation
    </NotificationType>
    <CausingGuarantee>ag2</CausingGuarantee>
    <SLAParameter>ResponseTimeThroughPutRatio
      TransactionRate</SLAParameter>
    </Action>
  </QualifiedAction>
  <ExecutionModality>Always</ExecutionModality>
</ActionGuarantee>

```

Figure 2.8: Action Guarantee example in WSLA

2.4 Implementation

2.4.1 Implementing Tools

For the purposes of our work, we have chosen to use some tools for the implementation, among many alternative solutions. The criteria for our selection were the connectivity and the efficient collaboration between them. These tools are:

- Sun GlassFish Application Server
- Eclipse
- WSLA plug-in for Eclipse
- PostgreSQL

Sun GlassFish Application Server



GlassFish Application Server ¹ is an open-source application server implementation of Java EE 5. In project GlassFish, Web services are first-class objects that can be easily monitored and managed.

GlassFish can track and graphically display operational statistics, such as the number of requests per second, the average response time and the throughput. One can enable monitoring for each of the Web services within an application and set the monitoring level to one of the following:

- **HIGH** — GlassFish monitors the response times, throughput, the total number of requests, the faults, and the details of the SOAP message trace.
- **LOW** — GlassFish monitors only the response times, throughput, the total number of requests, and the faults.
- **OFF** — GlassFish collects no monitoring data.

¹ <https://glassfish.dev.java.net/>

If monitoring is on for a Web service, it applies to all the operations in that Web service.



Eclipse and WSLA plug-in for eclipse

Eclipse ² is a free open-source Java program environment, using a custom user interface toolkit that runs on all platforms that supports Java 2. Eclipse requires a Java 2 runtime, so you need to install the Java 2 SDK first before installing Eclipse. It provides a powerful and feature rich integrated development environment (IDE) for Java.

There's an active community of third party Eclipse plug-in developers, both open source and commercial. As an Eclipse user, you're regularly rewarded with great new features from official Eclipse releases and from the plug-in development community. Such a plug-in is the WSLA plug-in ³ for eclipse developed by SOA-Blog.net. It is a free Eclipse plug-in that adds Web Service Level Agreement (WSLA) Language Support to the Eclipse platform. This tool is very useful as it provides a graphical interface to create WSLA documents with all its elements, including the Parties, the SLA Parameters and the Obligations.

The **WSLA plug-in** itself can be grouped into five main components:

- **WSLA Core Model:** provides access to WSLA model objects by mapping the XML document to a semantic model.
- **WSLA Multipage Editor:** a user-friendly form based editor to modify service level agreements that hides the complexity of XML. However, the editor also allows you to edit the XML source code directly, providing advanced editing capabilities such as syntax highlighting and outline views.

² <http://www.eclipse.org/>

³ <http://soa-blog.net/index.php?/archives/29-WebService-Level-Agreement-WSLA-Eclipse-Plugin.html>

- **WSLA Report Generator:** service level agreements reports can be printed out for review or audition from an WSLA model. This is especially beneficial to communicate SLAs with non technical users.
- **WSLA Wizards:** assist in the creation and management of WSLA documents.
- **WSLA Help System:** a comprehensive and extensive help system familiarizes with the usage of the plug-in. The help system can be accessed via the packaged eclipse help, the invocation of context sensitive help or online by visiting the WSLA InformationCenter. In addition, the entire WSLA specification is included as a reference in the help system.

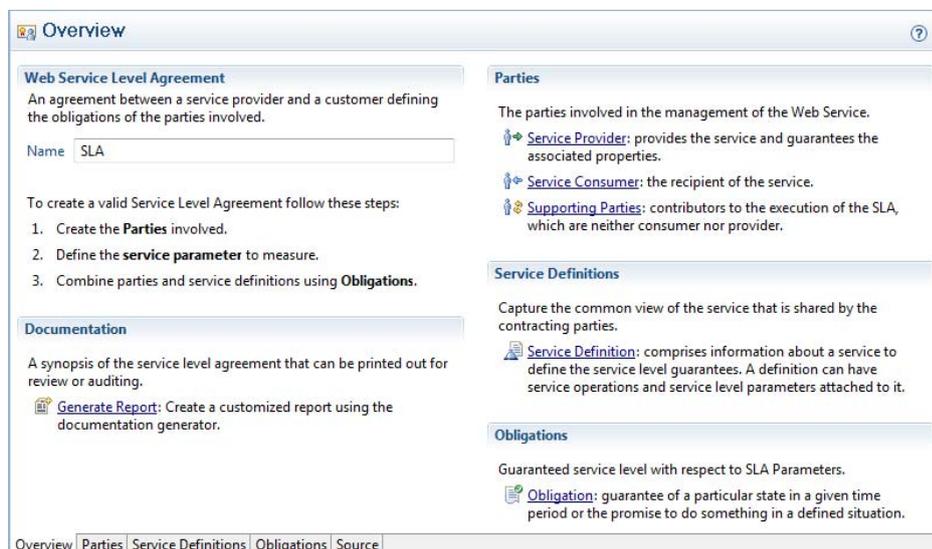


Figure 2.9: Main window of the WSLA plug-in for eclipse

Figure 2.10 shows a sample WSLA document, as extracted by the WSLA plug-in of eclipse. It is like an XML-editor, which discriminates with different colors the elements, the values and the types.

```
new_file.wsla | sample-outsourced.wsla
22<Parties>
23  <ServiceProvider
24    name="ACMEProvider">
25    <Contact>
26      <Street>PO BOX 218</Street>
27      <City>Yorktown, NY 10598, USA</City>
28    </Contact>
29    <Action xsi:type="WSDLSOAPOperationDescriptionType"
30      name="notification"
31      partyName="ZAuditing">
32      <WSDLFile>Notification.wsdl</WSDLFile>
33      <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
34      <SOAPOperationName>Notify</SOAPOperationName>
35    </Action>
36  </ServiceProvider>
37
38  <ServiceConsumer
39    name="XInc">
40    <Contact>
41      <Street>19 Skyline Drive</Street>
42      <City>Hawthorne, NY 10532, USA</City>
43    </Contact>
44    <Action xsi:type="WSDLSOAPOperationDescriptionType"
45      name="notification"
46      partyName="ZAuditing">
```

Figure 2.10: A sample WSLA document extracted from the eclipse plug-in



PostgreSQL⁴ is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures in multiple languages. It includes most SQL92 and SQL99 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, among others, and exceptional documentation.

It also provides a graphical interface, called pgAdmin. pgAdmin is a graphical front-end administration tool for PostgreSQL, which is supported on most popular computer platforms. The program is available in more than a dozen of

⁴ <http://www.postgresql.org/>

languages, and is free software released under the Artistic License. The stable release (named pgAdmin II) was first released on 16 January 2002.

2.4.2 Implementation procedure

In this section we analyze the implementation procedure of our work. It consists of six steps that are given in detail below:

1. Creation of WSLA documents using WSLA Plug-in for Eclipse.
2. Parsing of the WSLA document.
3. Storing of the ServiceLevelObjectives in a PostgreSQL database.
4. Execution of the Web Service using Sun GlassFish Application Server and extraction of the available metrics.
5. Computing of further metrics using the metrics, provided by GlassFish.
6. Comparing the agreed metrics values of SLA with the metrics extracted by the monitoring system

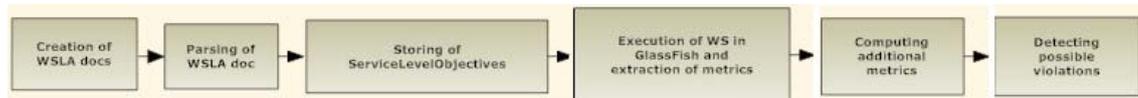


Figure 2.11: Steps of the implementation procedure

Creation of WSLA documents using WSLA Plug-in for Eclipse

For the purposes of our work it is essential to create WSLA documents, as it is difficult to have the real SLA documents the provider signs with the client. They are usually private documents that are not publicly available. For this reason, we used a WSLA plug-in for eclipse to create our WSLA documents.

Parsing of the WSLA document

As described in the previous section, a WSLA document has many elements, such as parties, service definition, obligations etc. From all of these, we are interested in the obligations, that define the metrics values that were agreed between the service provider and the client. In order to collect this information, we made a parser that is

basically an XML parser. This parser collects the following elements of the WSLA document:

- The Web service name
- The obliged party
- The start of validity period
- The end of validity period
- The predicate that applies for the specific value
- The name of the SLA parameter
- The agreed value of the SLA parameter and
- The EvaluationEvent

Storing of the ServiceLevelObjectives in a PostgreSQL database

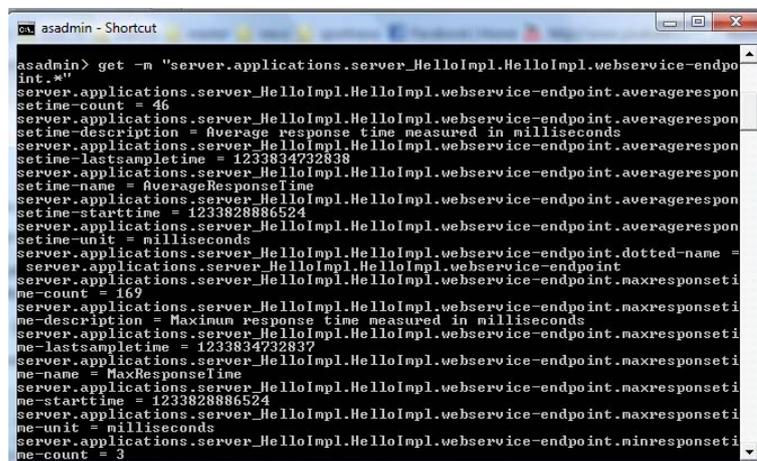
The previous collected elements must be saved permanently, in order to be used later. For this reason, we have used a PostgreSQL database to save them. In this database, we have created a table with nine columns (plus one for the ID), each of which store the corresponding value. The web service name, the obliged party, the predicate, the SLA parameter name and the Evaluation Event are stored as text values, the start and validity period as date values and the agreed value of the SLA Parameter as a numeric value.

Output pane									
Data Output									
Explain Messages History									
	ID integer	WSName text	Obliged text	Start date	End date	Predicate text	SLAParameter text	Value numeric	EvaluationEvent text
1	8	HelloImpl	ACMEProvider	2001-11-30	2001-12-31	Less	OverloadPercentage	0.3	NewValue
2	9	HelloImpl	ACMEProvider	2001-11-30	2001-12-31	Greater	Availability_UpTimeRatio	0.97	NewValue
3	10	HelloImpl	ACMEProvider	2001-11-30	2001-12-31	Less	Availability_CurrentDownr	10	NewValue

Figure 2.12: WSLA elements stored in the PostgreSQL database

Execution of the Web Service on Sun GlassFish Application Server and extraction of the available metrics.

As described in the tools subchapter, Sun GlassFish Application Server is a very useful tool that allows us to manipulate web services uploaded on it. We use the monitoring feature to collect all the metrics that are available. The metrics values are also depicted with graphs along with the time (figure 2.15), but we use the asadmin console to collect them via command line. As mentioned below, there are three levels of monitoring in Glassfish. We set the monitoring level to HIGH, in order to have as many metrics as possible. Below (figure 2.14), is the output of the statistics results of the running web service in the asadmin console.



```
asadmin> get -n "server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.*"
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.averageresponse-time-count = 46
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.averageresponse-time-description = Average response time measured in milliseconds
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.averageresponse-time-lastsampletime = 1233834732838
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.averageresponse-time-name = AverageResponseTime
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.averageresponse-time-starttime = 1233828886524
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.averageresponse-time-unit = milliseconds
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.dotted-name = server.applications.server_HelloImpl.HelloImpl.webservice-endpoint
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.maxresponse-time-count = 169
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.maxresponse-time-description = Maximum response time measured in milliseconds
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.maxresponse-time-lastsampletime = 1233834732837
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.maxresponse-time-name = MaxResponseTime
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.maxresponse-time-starttime = 1233828886524
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.maxresponse-time-unit = milliseconds
server.applications.server_HelloImpl.HelloImpl.webservice-endpoint.minresponse-time-count = 3
```

Figure 2.13: Metrics values in asadmin console

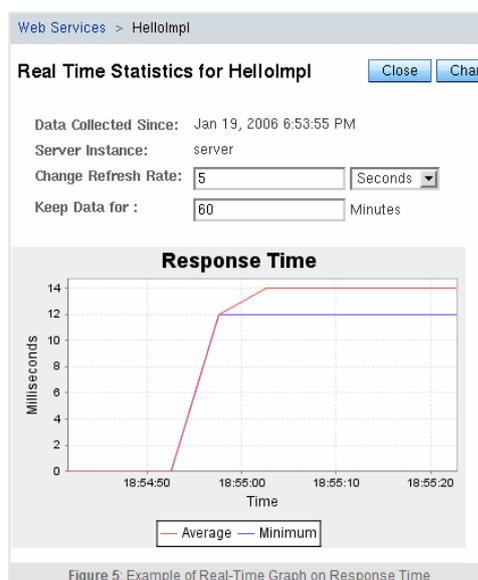


Figure 2.14: Graphical depiction of the Response Time

The command line results are more detailed, as they also provide composed metrics, such as min response time or average response time. Specifically, the extracted metrics values are:

- Average response time
- Min response time
- Max response time
- Response time
- Throughput
- Total number of authentication failures
- Total number of authentication successes
- Total number for requests that caused SOAP faults
- Total number of successful invocations of the method

Computing of further metrics using the metrics, providing by GlassFish

There are some other metrics that can be calculated using the above metrics, provided by GlassFish. These metrics are **availability** and **reliability**:

Availability is the probability that a service is up and running. It can be calculated in the following way:

$$availability = \frac{successfulInvocations}{TotalInvocations}$$

Reliability [28] is the Ratio of the number of error messages to total messages and can be calculated in the following way:

$$reliability = \frac{totalFaults}{totalMessages} \times 100$$

Comparing the agreed metrics values of SLA with the metrics extracted by the monitoring system

Now that we have the metrics values of the web service and the agreed values of the SLA, we can do the condition evaluation. Thus, we check if the web service running on the application server has a corresponding SLA with the agreed values stored in the database. If there is such a matching, we can compare the metric values to see and if there is any violation. Once the Condition Evaluation Service has determined that an SLO has been violated, corrective management actions need to be carried out. The Management Service will retrieve the appropriate actions to correct the problem, as specified in the SLA.

2.4.3 UML diagrams

The UML class diagram of our implementation is shown in Figure 2.14. The ServiceLevelObjectiveClass implements objects of the corresponding type. It defines all the fields of the ServiceLevelObjective as string type and contains all the appropriate set and get methods to handle them.

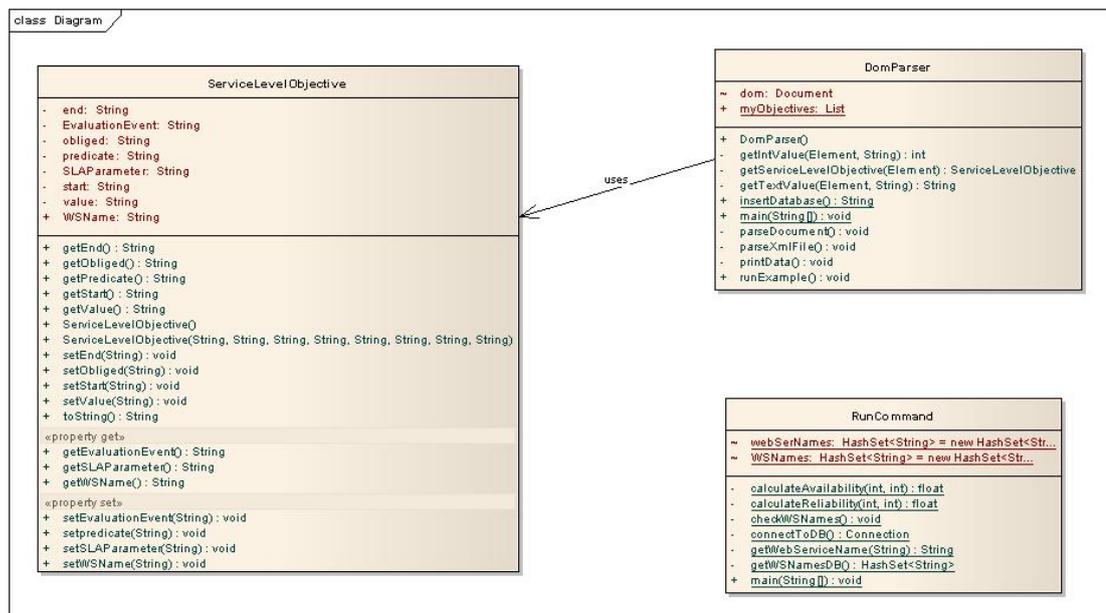


Figure 2.15: The UML class diagram of our implementation

The role of the DomParser class (step 2) is to parse the WSLA document. As we have already mentioned, WSLA is an XML-based language, so a simple XML parser is needed. Methods parseDocument() and parseXmlFile() are responsible for the parsing, getIntValue(), getServiceLevelObjective() and getTextValue() methods are used to get the values of each of the corresponding types and the printData() method prints the results (name and values of metrics), after we have inserted the parsed values in the PostGreSQL database with the insertDatabase() method.

The RunCommand class (steps 4-6) collects the metrics from the GlassFish ApplicationServer and computes additional metrics, such as availability and reliability. It checks the web service names to see if there is a corresponding WSLA document for the deployed Web services at the database and if there is such a correlation it checks for possible violations in all available metric values. An alert message is shown for every violation.

The UML sequence diagram is shown in Figure 2.15 and corresponds to steps 4-6 of the previous section. In the beginning, the user asks the *condition evaluator* if there are any violations in the metrics values. The *condition evaluator* follows four steps to complete the procedure.

1. It asks for and gets the metrics from the *Glassfish Application Server*.
2. The *metric computer*, after getting the retrieved metrics from *Glassfish*, computes some further metrics and returns them to the condition evaluator.
3. The *condition evaluator* gets the agreed metric values and the predicate from the WSLA database and
4. The *condition evaluator* computes possible violations and returns them to the user.

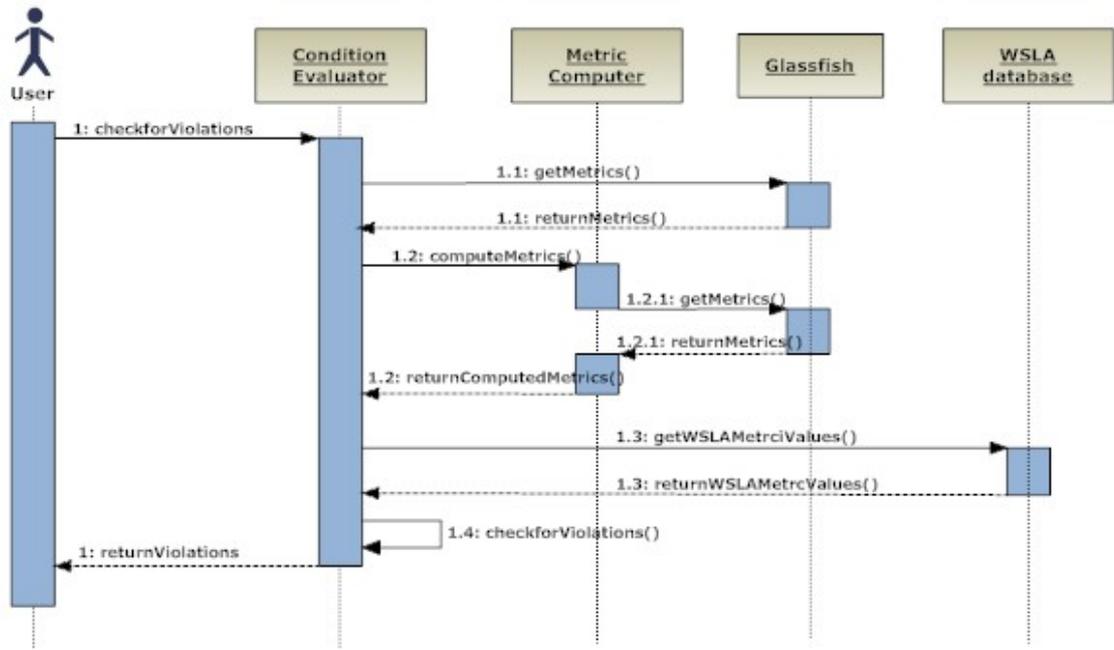


Figure 2.16: The UML sequence diagram

Chapter 3

3 Computing QoS Values of WS Compositions

3.1 Introduction

Enterprises can use a singular service to accomplish a specific business task, such as billing or inventory control. However, for enterprises to obtain the full benefit of Web services, business process and transactional-like Web services functionality is required that is well beyond that found in informational Web services. When enterprises need to compose several services together to create a business process such as customized ordering, customer support, procurement, and logistical support, they need to use complex Web services. Complex or composite services typically involve the assembly and invocation of many pre-existing services possibly found in diverse enterprises to complete a multi-step business interaction. Consider for example a supply-chain application involving order taking, stocking orders, sourcing, inventory control, financials, and logistics. Numerous document exchanges will occur in this process including requests for quotes, returned quotes, purchase order requests, purchase order confirmations, delivery information, and so on. Long-running transactions and asynchronous messaging will also occur, and business “conversation” and even negotiations may occur before the final agreements are reached. This functionality is a typical characteristic of business processes.

Complex Web services can in turn be categorized according to the way they compose simple services. Some complex Web services compose simple services that exhibit programmatic behavior whereas others compose services that exhibit mainly interactive behavior where input has to be supplied by the user. This makes it natural to distinguish between the following two types of Web services:

- Complex Web services that compose programmatic Web services: The clients of these Web services can assemble simple services, to build complex services. A typical example of a simple service exhibiting programmatic behavior could be an inventory checking service that comprises part of an inventory management process.
- Complex services that compose interactive Web services: These services expose the functionality of a Web application's presentation layer. They frequently expose a multi-step Web application behavior that combines a Web server, an application server, and underlying database systems and typically deliver the application directly to a browser and eventually to a human user for interaction. Clients of these Web services can incorporate interactive business processes into their Web applications, presenting integrated applications from external services providers. Obviously interactive services can be combined with programmatic services thus delivering business processes that combine typical business logic functionality with Web browser interactivity. [26]

3.2 Types of Web service composition

There are two types of Web service composition based on the distinction between syntactic and semantic Web services: *Static* and *Dynamic* Compositions.

3.2.1 Static Services Composition

A relevant feature for Web services is the mechanism for their reuse when complex tasks are carried out. Composition rules deal with how different services are composed into a coherent global service. In particular, they specify the order in which services are invoked, and the conditions under which a certain service may or may not be invoked. Two possible approaches are currently investigated for the static service composition.

The first approach, referred to as Web services *orchestration*, combines available services adding a central coordinator (the orchestrator) which is responsible

for invoking and combining the single sub-activities. Orchestration refers to an executable business process that may result in a long-lived, transactional, multi-step process model. With orchestration, the business process interactions are always controlled from the perspective of one of the business parties involved in the process.

The second approach, referred to as Web services *choreography*, does not assume the exploitation of a central coordinator but it defines complex tasks via the definition of the conversation that should be undertaken by each participant. Choreography tracks the sequence of messages that may involve multiple parties and multiple sources, including customers, suppliers, and partners, where each party involved in the process describes the part it plays in the interaction and no party “owns” the conversation.

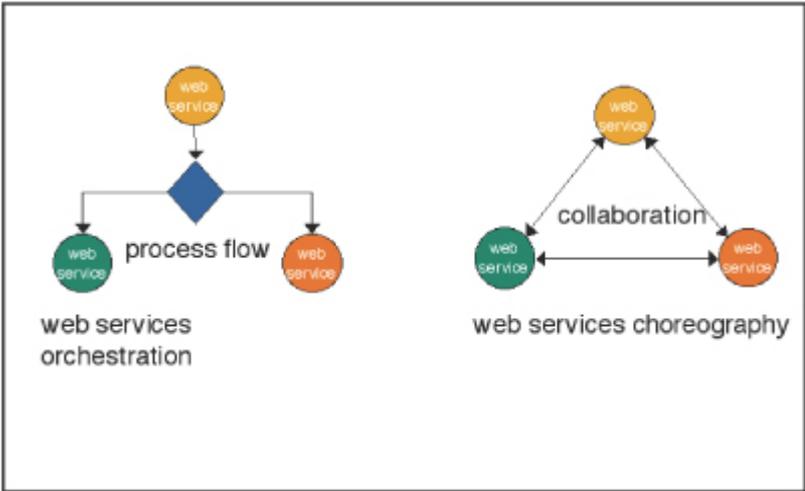


Figure 3.1: Orchestration and choreography

Choreography is more collaborative in nature than orchestration (figure 3.1). It is described from the perspectives of all parties, and, in essence, defines the shared state of the interactions between business entities. This common view can be used to determine specific deployment implementations for each individual entity. Choreography offers a means by which the rules of participation for collaboration can be clearly defined and agreed to, jointly. Each entity may then implement its portion of the choreography as determined by their common view.

3.2.2 Dynamic Services Composition

Web Services [7] are designed to provide interoperability between different applications. The platform and language independent interfaces of the web services allow the easy integration of heterogeneous systems. Web languages such as UDDI, WSDL and SOAP define standards for service discovery, description and messaging protocols. However, these web service standards do not deal with the dynamic composition of existing services. The new industry initiatives to address this issue such as BPEL4WS focus on representing composition where flow of the information and the binding between services are known a priori. A more challenging problem is to compose services dynamically. In particular, when a functionality that cannot be realized by the existing services is required, the existing services can be combined together to fulfill the request. The dynamic composition of services requires the location of services based on their capabilities and the recognition of those services that can be matched together to create a composition. The full automation of this process is still the object of ongoing research activity, but accomplishing this goal with a human controller as the decision mechanism can be achieved. The main problem for this goal is the gap between the concepts people use and the data computers interpret. This barrier can be overcome using semantic web technologies.

3.3 Web service composition patterns

Web services can be composed using different patterns. These patterns are based on the usual workflow patterns and are analyzed in this subsection. There are many situations where two or more of these patterns are combined to create a complex web service composition. We'll illustrate each of the patterns with a figure.

3.3.1 Sequence Pattern

The sequence pattern indicates that the web services are executed one after the other, not in parallel but in sequence. The sequential order is predefined and must be followed up in order to have a successful execution.

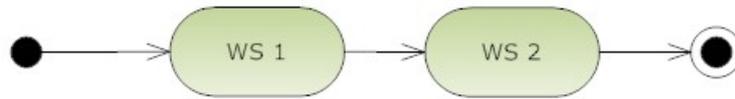


Figure 3.2: Sequence pattern

3.3.2 Parallel pattern

The parallel pattern (figure 3.3) indicates that two or more web services can be executed in parallel. This means that they are independent and the execution of WS2 does not affect the execution of WS3. The order in which they are processed is not defined, but in the end they are merged with synchronization.

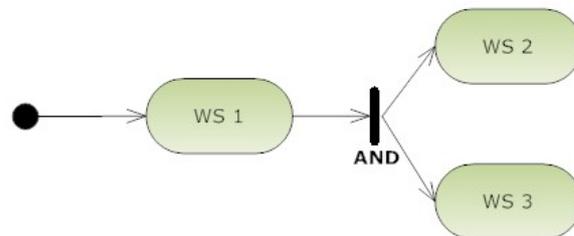


Figure 3.3: Parallel pattern

3.3.3 Synchronization pattern

The synchronization pattern (figure 3.4) indicates that the process will continue after the parallel pattern of the Web service is executed. This pattern is implemented mostly by the use of receive statements.

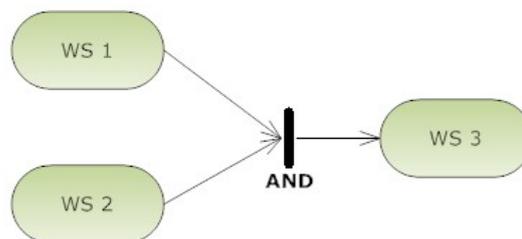


Figure 3.4: Synchronization pattern

3.3.4 Exclusive choice pattern

The exclusive choice pattern (figure 3.5) defines a point in the business workflow, where a certain condition based on a decision in the flow is taken. In fact a XOR condition is used.

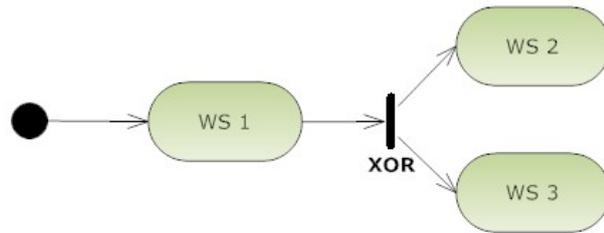


Figure 3.5: Exclusive choice pattern

3.3.5 Simple merge pattern

The simple merge pattern (figure 3.6) defines a point in the flow of execution, where two or more alternative branches are merged. It is important to mention that the simple merge pattern does not support any kind of synchronization, which means, that none of the alternative processes is ever executed in parallel.

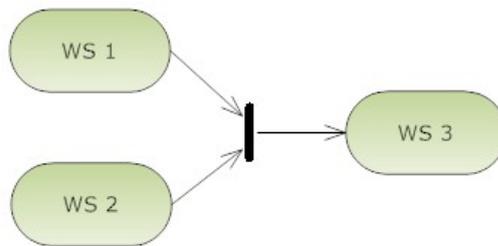


Figure 3.6: Simple merge pattern

3.3.6 Conditional pattern

The conditional pattern (figure 3.7) indicates that there are multiple activities (a_1, a_2, \dots, a_n) among which only one activity can be executed. As we'll describe later, each of these activities a_i have a p_i probability to be executed.

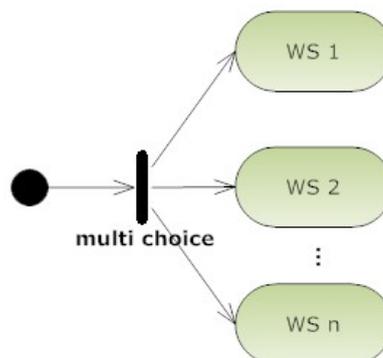


Figure 3.7: Conditional pattern

3.3.7 Synchronizing merge pattern

The synchronizing merge pattern (figure 3.8) marks a point in the process execution, where several branches merge into a single one. If one or more processes are active, the flow is triggered until these processes are finished.

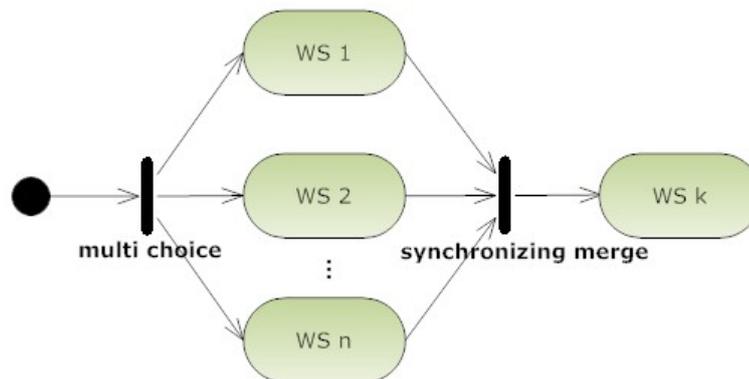


Figure 3.8: Synchronizing merge pattern

3.3.8 Multi-merge pattern

The multi-merge pattern (figure 3.9) joins two or more different workflows without synchronization together. This means that results, processed on different paths, are passed to other activities in the order in which they are received.

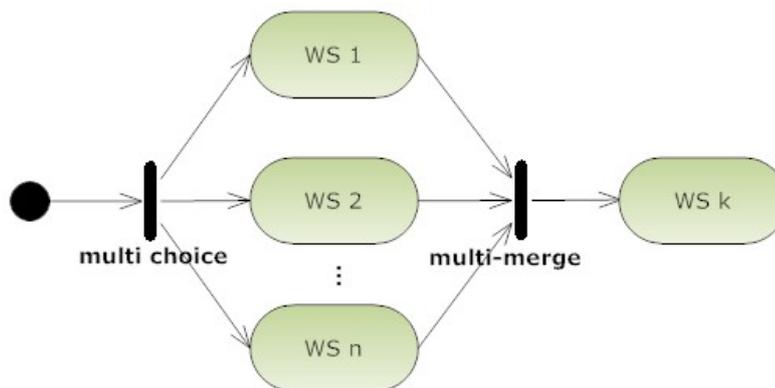


Figure 3.9: Multi-merge pattern

3.3.9 Loop pattern

The loop pattern (figure 3.10) indicates that a certain point in the composition block is executed repeatedly. There have to be no restrictions on the number, location, and nesting of these points. In our examples we use the while loop.

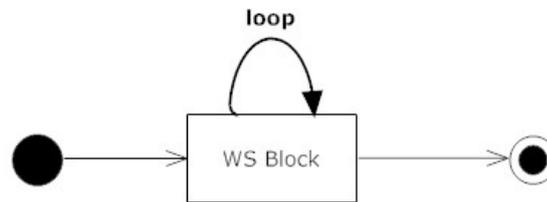


Figure 3.10: Loop pattern

3.3.10 Deferred choice pattern

The deferred choice pattern (figure 3.11) describes a point in a process where some information is used to choose one among several alternative branches. This information is not necessarily available when this point is reached.

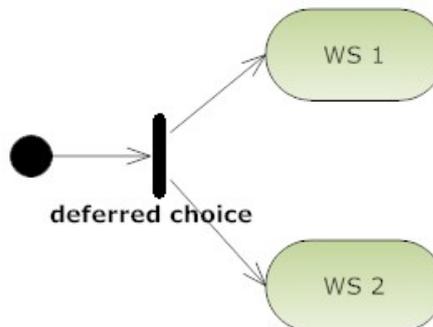


Figure 3.11: Deferred choice pattern

3.3.11 Interleaved Parallel Routing

In this pattern, a collection of activities is executed in an arbitrary order (figure 3.12). Each activity in the collection is executed once and the order between the activities is decided at run-time. In any case, no two activities in the collection can be active at the same time.

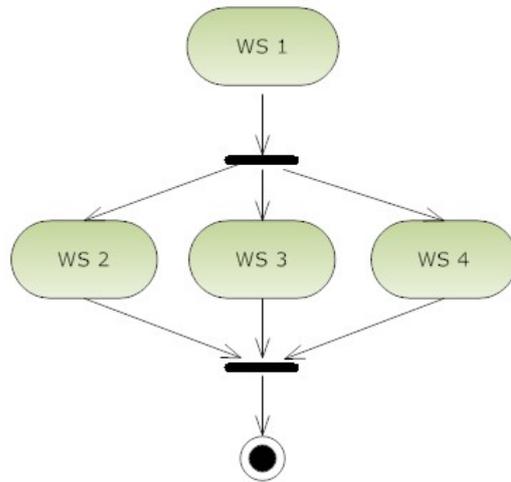


Figure 3.12: Interleaved parallel routing

3.3.12 Milestone pattern

This pattern [25] describes a situation in which a certain activity can only be enabled if a milestone has been reached which has not yet expired (figure 3.13). A milestone is a point in the process where a given activity A has finished and a subsequent activity B has not yet started. For example, this is the case of a student which can cancel a subject at any time after the semester has begun and prior to the first exam.

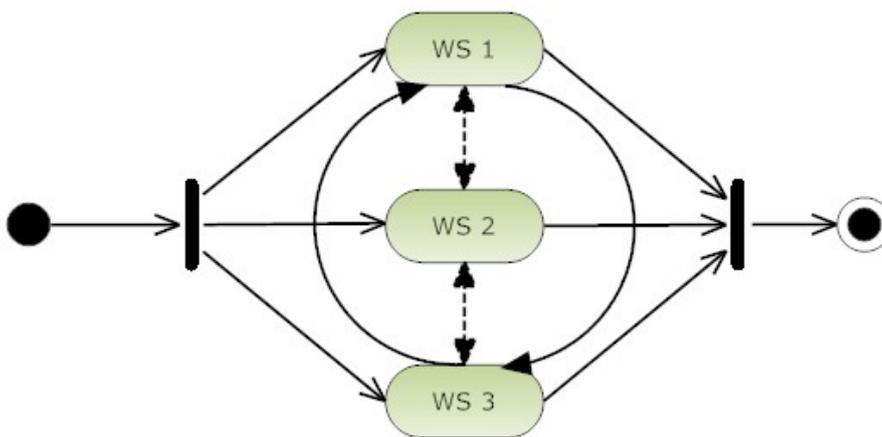


Figure 3.13: Milestone pattern

3.4 Computing basic metrics for composed services

In this section we study the problem of computing some basic metrics for composed web services, having available the corresponding values of the constituent simple services. As it is difficult and inaccurate to define formulas for all the metrics mentioned in chapter 1, we have made an effort to calculate the most commonly used metrics for as many as possible composition patterns. These metrics are: response time, throughput, reliability and cost, defined as $RT(s)$, $T(s)$, $R(s)$ and $C(s)$ respectively. The constituent web services are denoted as s_1, s_2, \dots, s_n and the web service composition that includes these services as $w(s_1, s_2, \dots, s_n)$. For the conditional pattern we denote as p_i the probability of a service s_i to be selected. Finally we denote as $SO(s_i, p_i)$ the selection operation for the conditional patterns, which selects the service s_i with execution probability p_i .

3.4.1 Computing response time

The response time metric compute the time a service provider needs to serve a client request. It is usually defined in milliseconds and is the most commonly used metric for web services.

For the sequential pattern, the response time is defined as the sum of the response times of the constituent web services:

$$RT(s)_{\text{sequential}} = \sum_{i=1}^n RT(s_i)$$

For the parallel, the synchronization and the simple merge pattern, the response time of the compositions is defined as the maximum response time of the constituent web services.

$$RT(s)_{\text{parallel}} = \max\{RT(s_i)\}$$

$$RT(s)_{\text{synchronization}} = \max\{RT(s_i)\}$$

$$RT(s)_{simple_merge} = \max\{RT(s_i)\}$$

For the exclusive choice and the deferred choice pattern, the response time is calculated by the selection operation, which selects one of the n possible Web services. In particular it is equal to the response time of the selected web service and is defined as:

$$RT(s)_{exclusive_choice} = RT(SO(s_i, p_i))$$

$$RT(s)_{deferred_choice} = RT(SO(s_i, p_i))$$

For the conditional and the synchronizing merge pattern, the response time is defined as the maximum response time of the selected web services, chosen by the selection operation.

$$RT(s)_{conditional} = \max_i(RT(SO(s_i, p_i)))$$

$$RT(s)_{synchronizing_merge} = \max_i(RT(SO(s_i, p_i)))$$

Finally, the response time of the loop pattern is n times the response time of the loop, where n is the number of iterations.

$$RT(s)_{loop} = n \times RT_{(loop)}$$

3.4.2 Computing throughput

Throughput computes the number of instances completed per unit time, usually per second. It is a very useful metric as it measures the performance of a web service. For composed web services, throughput can be extracted if we have available the throughputs of the constituent web services.

For the sequential pattern, if we have n constituent web services, each of them executed only once, the throughput of this composed service is defined from the type:

$$T(s)_{sequential} = \frac{1}{\sum_{i=1}^n \frac{1}{T(s_i)}}$$

Proof: Suppose that we have three web services, with throughputs x req/s, y req/s and z req/s, respectively. Then, for one invocation of the first web service $1/x$ seconds are needed. Accordingly, $1/y$ seconds and $1/z$ seconds lasts one invocation of the second and third web service. If we compose these web services sequentially, we need $(1/x + 1/y + 1/z)$ seconds. So, the throughput of the composed service is the inverse of this sum.

If we have web services, that are executed in parallel, the throughput of the composed web service is equal to the minimum of the constituent services' throughputs. This means that in order for the composed service to increase the overall throughput, it would need to optimize the service with the lowest throughput. So, the throughput's type, for the parallel, the synchronization and the simple merge patterns is:

$$T(s)_{\text{parallel}} = \min_i T(s_i)$$

$$T(s)_{\text{synchronization}} = \min_i T(s_i)$$

$$T(s)_{\text{simple_merge}} = \min_i T(s_i)$$

For the exclusive choice and the deferred choice patterns, the throughput is equal to the throughput of the selected web service.

$$T(s)_{\text{exclusive_choice}} = T(SO(s_i, p_i))$$

$$T(s)_{\text{deferred_choice}} = T(SO(s_i, p_i))$$

For the multi-choice/conditional and the synchronizing merge patterns, the throughput is equal to the minimum throughput of the selected web services.

$$T(s)_{\text{conditional}} = \min_i T(SO(s_i, p_i))$$

$$T(s)_{\text{synchronizing_merge}} = \min_i T(SO(s_i, p_i))$$

Finally, the throughput of a composed web service using the pattern loop with n iterations is defined by the type:

$$T(s) = \frac{1}{\sum_{i=1}^n \frac{1}{T_{(loop)}}}$$

3.4.3 Computing reliability

As described in the first chapter reliability represents the ability of a service to function correctly and consistently and is usually expressed in terms of number of transactional failures per month or year. In our work we define reliability as the probability that the service can be successfully completed.

For the sequential, parallel, synchronization, simple merge and loop pattern, reliability is defined as the product of the reliabilities of the constituent web services. This happens because in all these situations all the services of the composition are executed. For example, if we have a composition of three web services s_1 , s_2 , s_3 , following one of the above mentioned patterns, with probability 70%, 80% and 90% to be completed successfully, respectively, the overall reliability of the composed service is approximately 50%.

$$R(s)_{sequential} = \prod_{i=1}^n R(s_i)$$

$$R(s)_{parallel} = \prod_{i=1}^n R(s_i)$$

$$R(s)_{synchronization} = \prod_{i=1}^n R(s_i)$$

$$R(s)_{simple_merge} = \prod_{i=1}^n R(s_i)$$

For the loop pattern the availability is equal to the loop's availability powered to n , where n is the number of iterations.

$$R(s)_{loop} = R_{loop}^n$$

For the exclusive choice and the deferred choice pattern, the overall reliability is defined by the reliability of the selected web service. The selection operation SO is used once again for this reason.

$$R(s)_{\text{exclusive choice}} = R(SO(s_i, p_i))$$

$$R(s)_{\text{deferred choice}} = R(SO(s_i, p_i))$$

Finally, the reliability for multi-choice/conditional and synchronizing merge patterns depends on the reliability of the selected services. Thus, it is equal to the product of the n corresponding reliabilities.

$$R(s)_{\text{multi-choice}} = \prod_{i=1}^n R(SO(s_i, p_i))$$

$$R(s)_{\text{synchronizing merge}} = \prod_{i=1}^n R(SO(s_i, p_i))$$

3.4.4 Computing cost

Computing cost is defined the amount of money paid for the composed service. It is obvious, that the cost is equal to the sum of costs of the n constituent web services, except for the case of conditional patterns where it is exactly the same with the cost of the selected service. Thus, the cost for sequential, parallel and synchronization patterns is:

$$C(s) = \sum_{i=1}^n C(s_i)$$

For the conditional and synchronizing merge patterns, the cost is:

$$C(s) = \sum_{i=1}^n C(SO(s_i, p_i))$$

For the loop pattern, if we have n iterations, the cost is:

$$C(s) = n \times C(\text{loop})$$

For the exclusive choice and deferred choice patterns, the cost is:

$$C(s) = C(SO(s_i, p_i))$$

3.5 Summarizing the results

In table 3.1, we summarize the results of the previous section. As we can see, there are three groups of composition patters that have the same metric types. These groups are: exclusive choice – deferred choice, multi-choice – synchronizing merge and parallel – synchronization – simple merge. The sequence pattern has different types for the response time and throughput metric and the same reliability and cost types with the other patterns, while the loop pattern is differentiated in all the metrics from the other patterns.

Metric Pattern	Response Time	Throughput	Reliability	Cost
Sequence	$\sum_{i=1}^n RT(s_i)$	$\frac{1}{\sum_{i=1}^n \frac{1}{T(s_i)}}$	$\prod_{i=1}^n R(s_i)$	$\sum_{i=1}^n C(s_i)$
Parallel	$\max\{RT(s_i)\}$	$\min_i T(s_i)$	$\prod_{i=1}^n R(s_i)$	$\sum_{i=1}^n C(s_i)$
Synchronization	$\max\{RT(s_i)\}$	$\min_i T(s_i)$	$\prod_{i=1}^n R(s_i)$	$\sum_{i=1}^n C(s_i)$
Exclusive choice	$RT(SO(s_i, p_i))$	$T(SO(s_i, p_i))$	$R(SO(s_i, p_i))$	$C(SO(s_i, p_i))$
Simple merge	$\max\{RT(s_i)\}$	$\min_i T(s_i)$	$\prod_{i=1}^n R(s_i)$	$\sum_{i=1}^n C(s_i)$
Multi-choice/ conditional	$\max_i RT(SO(s_i, p_i))$	$\min_i T(SO(s_i, p_i))$	$\prod_{i=1}^n R(SO(s_i, p_i))$	$\sum_{i=1}^n C(SO(s_i, p_i))$
Synchronizing merge	$\max_i RT(SO(s_i, p_i))$	$\min_i T(SO(s_i, p_i))$	$\prod_{i=1}^n R(SO(s_i, p_i))$	$\sum_{i=1}^n C(SO(s_i, p_i))$
Loop	$n \times RT_{(loop)}$	$T(s) = \frac{1}{\sum_{i=1}^n \frac{1}{T_{(loop)}}}$	R_{loop}^n	$n \times C_{(loop)}$
Deferred choice	$RT(SO(s_i, p_i))$	$T(SO(s_i, p_i))$	$R(SO(s_i, p_i))$	$C(SO(s_i, p_i))$

Table 3.1: Metric Types for composed Web services

3.6 Use case

In this section, we provide a use case example to exemplify how we can compute the previous metrics of a composed service that combines many different patterns. We assume that we have a composed web service that simulates the use of a calculator and we want the result of the following mathematical operation:

$$\frac{\left(\frac{x}{y}\right) + ((x+y) + (x-y) + (x \times y)) \times \left(\frac{x}{y}\right)^3 + ((x+y) - \left(\frac{x}{y}\right))^2}{y}$$

This operation can be described with a UML sequence diagram:

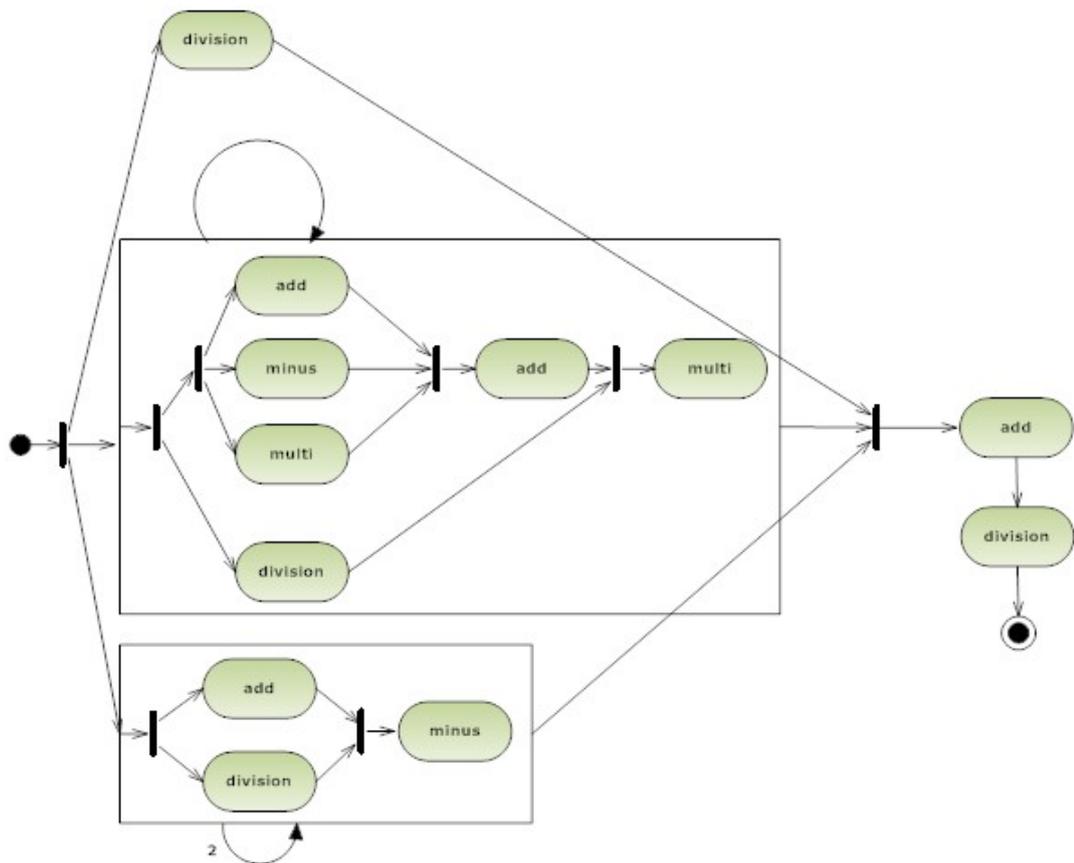


Figure 3.14: UML activity diagram for use case

As we can see from the diagram, this complex service combines sequence and synchronization patterns. Specifically, the second block of operations includes two synchronizations patterns and two sequence patterns, while the third block includes one synchronization pattern and one sequence pattern. The power to 3 and

power to 2 aren't loop operations, since we already know the result of each block. Thus, the power to three of the second block means that we have two more multiplication operations ($RT_{(block)}^3$). Let's now compute the metrics described in the previous section having available the metrics of the constituent services, as described in table 3.2:

Metric Operation	Response Time	Throughput	Reliability	Cost
Add	10ms	40req/s	98%	5
Minus	10ms	40req/s	98%	5
Multi	40ms	10req/s	95%	20
Division	40ms	10req/s	95%	20

Table 3.2: Metric values for constituent Web services (use case)

Response time: To compute the response time of this composed web service we start from the inner operations. To make the calculation clearer we use the same UML sequence diagram with the evolution of the response time at the right corner of each constituent operation.

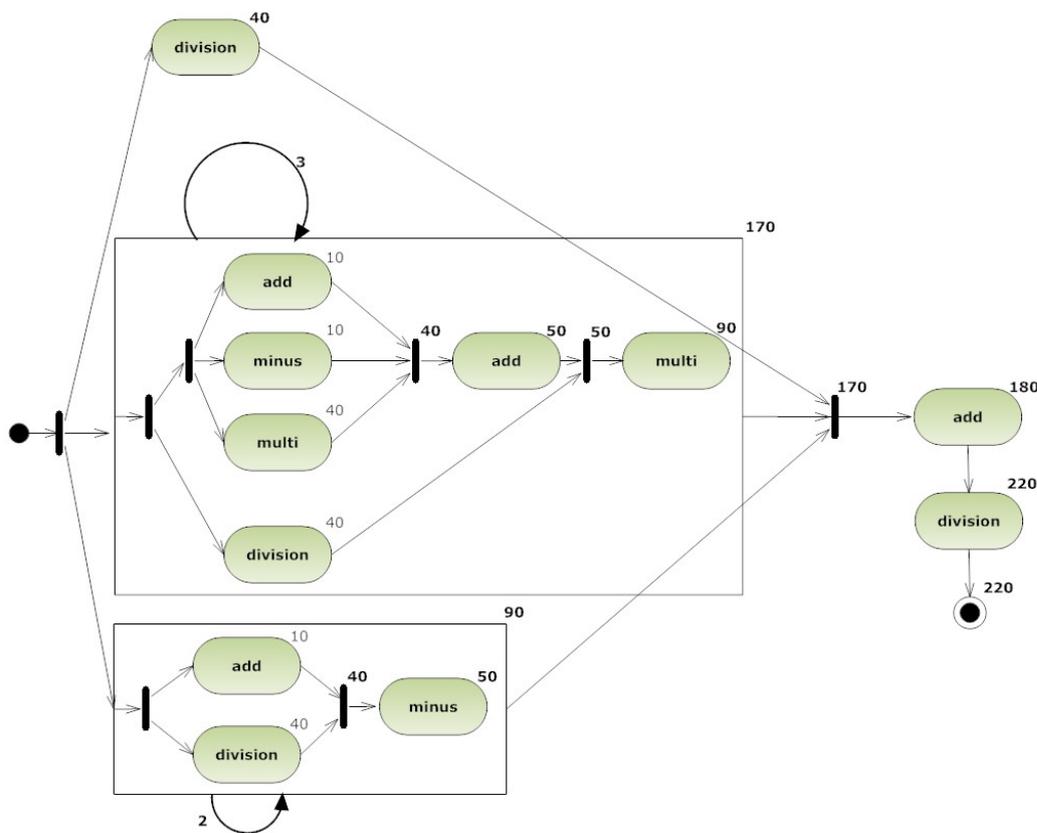


Figure 3.15: Response time activity diagram (use case)

As we can see, the response time of the composed service that computes the complex mathematical operation is 220ms.

Throughput: With the same descriptive way, we calculate the throughput of the composed service.

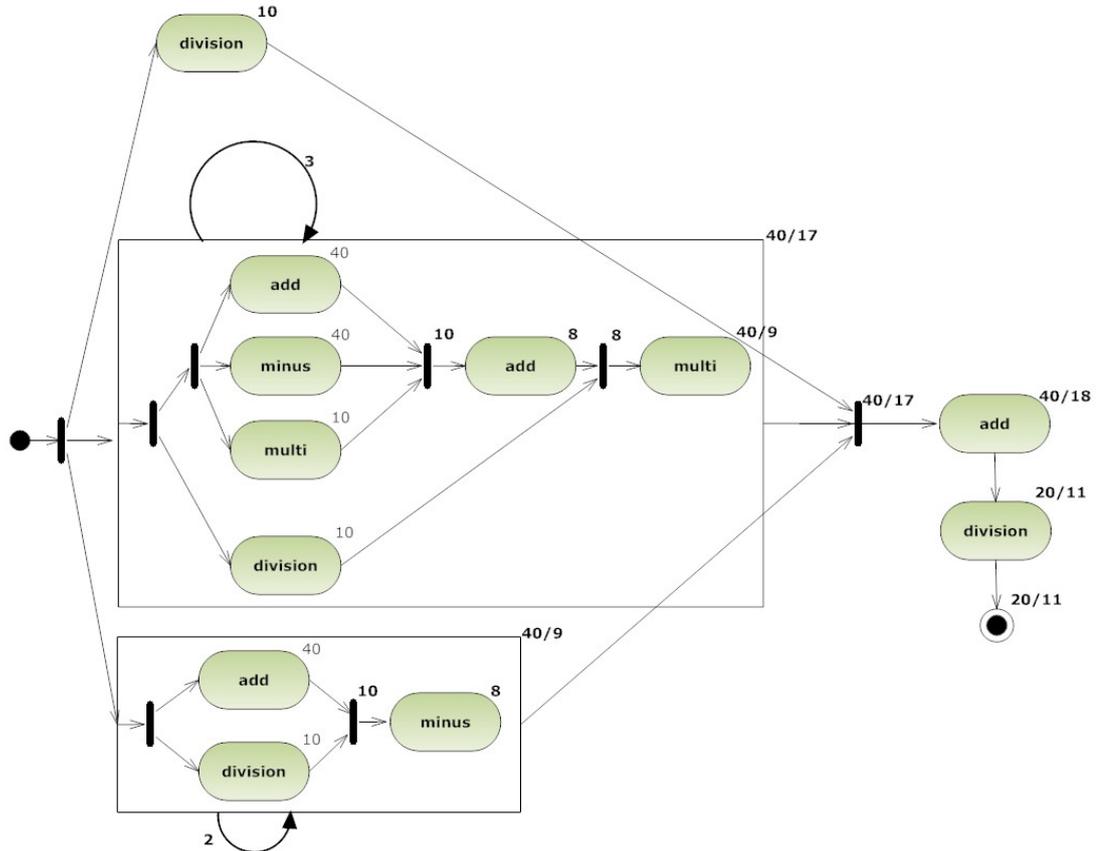


Figure 3.16: Throughput activity diagram (use case)

As we can see, the throughput of the composed service that computes the complex mathematical operation is $20/11 \approx 1,8req/sec$. To exemplify the calculation, below we prove the throughput value of the second block.

The throughput of the first inner synchronization pattern is:

$$T(s) = \min\{40,40,10\} = 10req/sec$$

Then, an add operation is executed in sequence:

$$T(s) = \frac{1}{\frac{1}{10} + \frac{1}{40}} = 8req/sec$$

This addition is executed in parallel with the division operation:

$$T(s) = \min\{10,8\} = 8req/sec$$

Then, a multiplication activity is executed in sequence:

$$T(s) = \frac{1}{\frac{1}{8} + \frac{1}{10}} = \frac{40}{9}req/sec$$

Finally, two multiplications operations take place to execute the power to 3 operation:

$$T(s) = \frac{1}{\frac{\frac{1}{40} + \frac{1}{10}}{9}} = \frac{40}{13}req/sec$$

$$T(s) = \frac{1}{\frac{\frac{1}{40} + \frac{1}{10}}{\frac{13}{13}}} = \frac{40}{17}req/sec$$

Reliability: The UML diagram below shows how we compute the reliability value of our composed service:

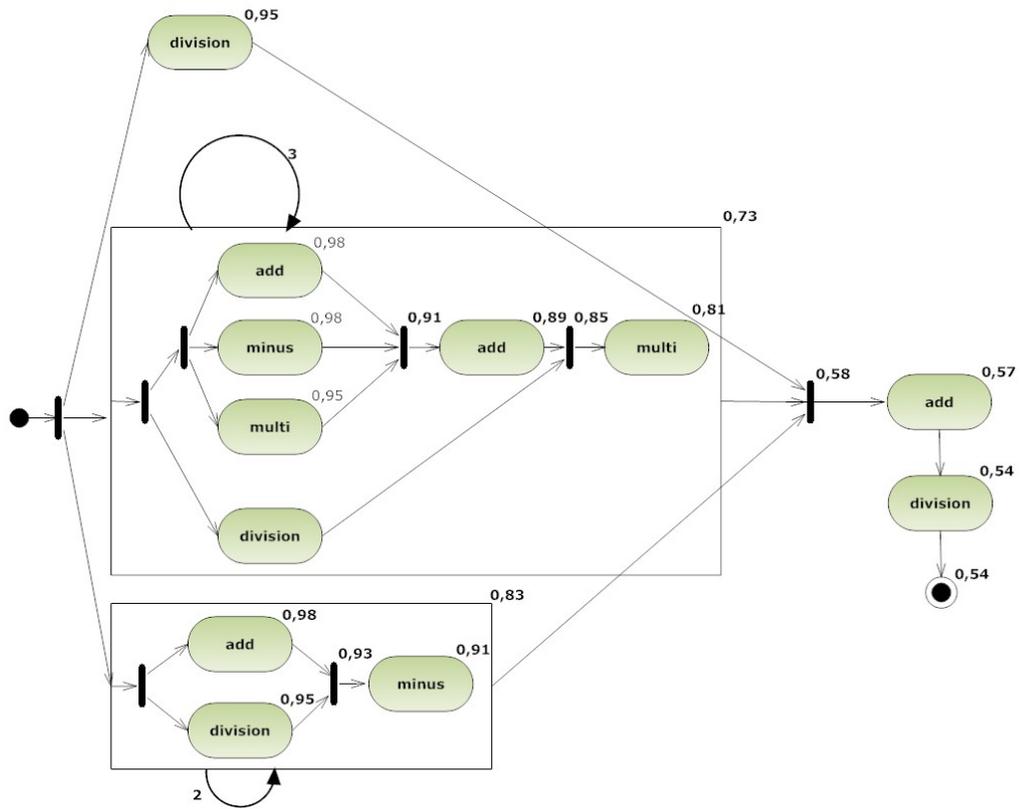


Figure 3.17: Reliability activity diagram (use case)

The reliability of the composed service is 54%. That means that the probability to be executed successfully is 54%. This value is the result of the multiplication of the availability values of all the constituent services.

Cost: Finally, we compute the cost of the composed service, which is in fact the amount of money that the client has to pay in order to use this service. The following diagram shows the evolution of the cost value.

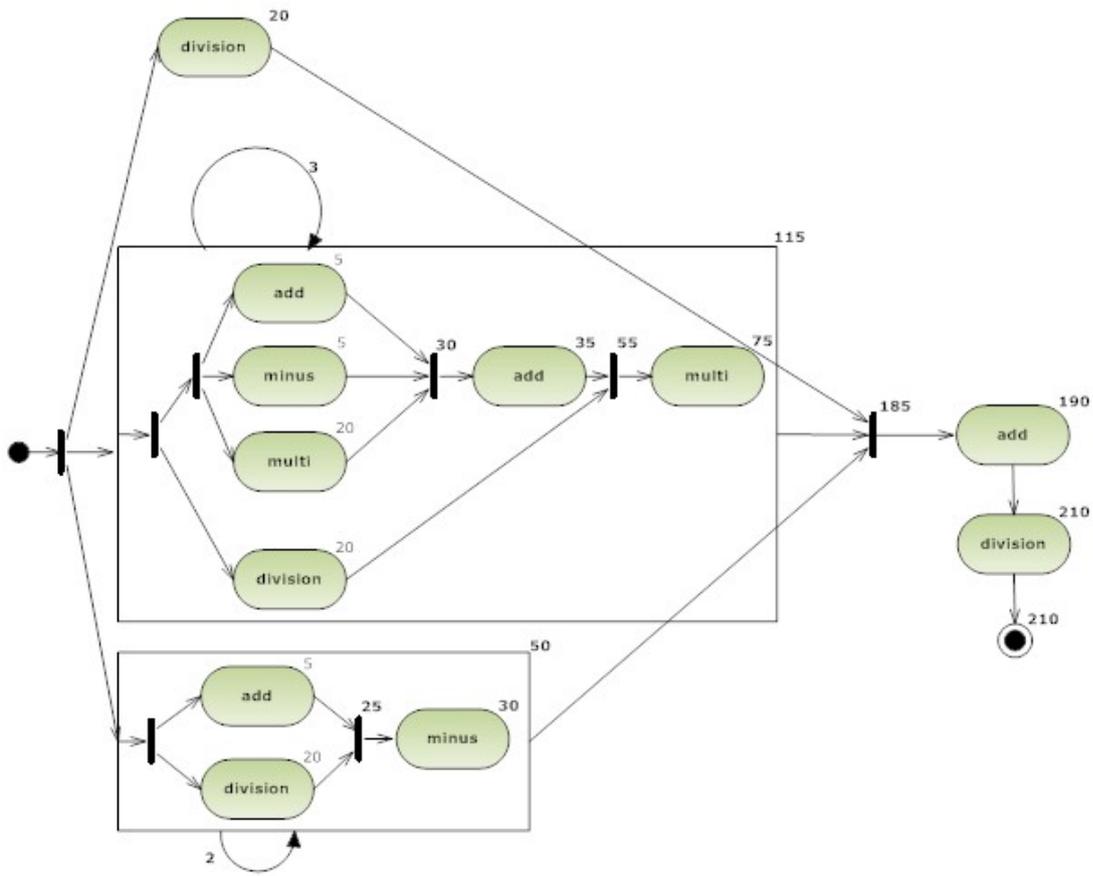


Figure 3.18: Cost activity diagram (use case)

The cost of our composed service is 210 units. This value is the result of the addition of the costs of all the constituent services.

Chapter 4

4 Experimental Evaluation

4.1 Introduction

In this Section we experimentally measure some basic metrics of web services. We provide diagrams that depict the performance of the services. It is worth mentioning that our experimental findings confirm to a great extent the theoretical analysis of the previous chapter.

To provide a representative overview on the QoS values, we monitored the Web services for over 10 hours, while constantly evaluating all parameters. Our Web services were implemented with Java and the composition was fulfilled in the ActiveBPEL environment.

All experiments were carried out in a PC with processor Intel Core2 Duo 1,80 GHz, 2 GB Ram, running Windows Vista 32-bit. Also, time was measured in milliseconds and the throughput in requests per second. We assumed that the cost of a web service is measured in Euros for our experiments.

4.2 Experiments

The first experiment we have performed is measuring the time in connection with the number of BPEL requests (figure 4.1). The Web service we use is an addition Web service, which simply adds two integers. We start the experiments from five requests, and we scale up to 25 activities, corresponding to a service that performs the addition operation in equal time.

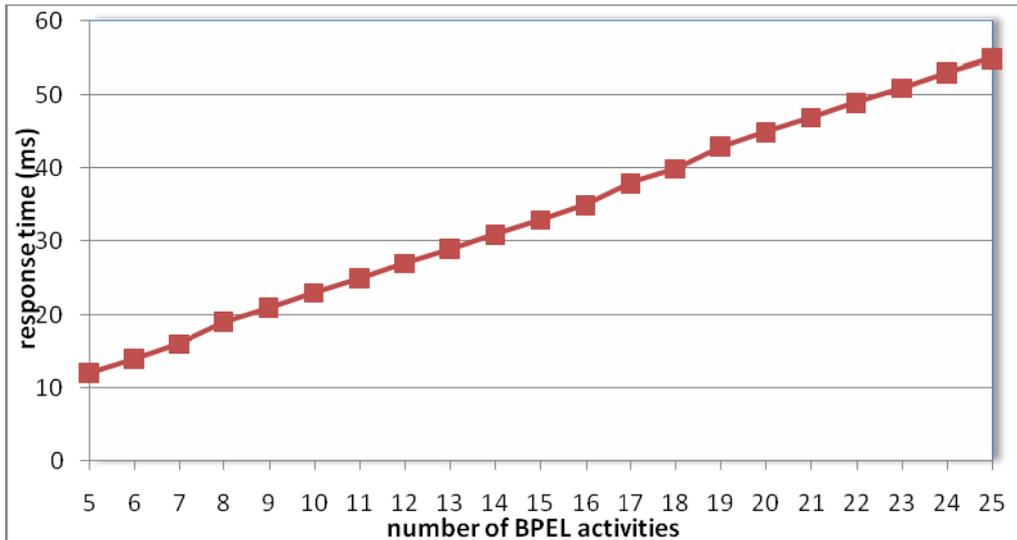


Figure 4.1: Response Time and number of BPEL activities

As expected, the response time is increasing accordingly to the number of BPEL activities. For example, the case of 5 activities takes 12ms, for 15 activities the response time is 33ms and for 25 requests it is 55ms.

In figure 4.2 we report the results of our second experiment. We are running a simple service on the Glassfish application server and we measure its response time for consecutive executions. As we can see, only for the first execution the response time is 20 ms, while for all others it fluctuates from 1ms to 4ms. This behavior may be justified by the server's instantiation of the service. We must mention that nothing else was running in the background, except the application server, so as to affect the service's execution.

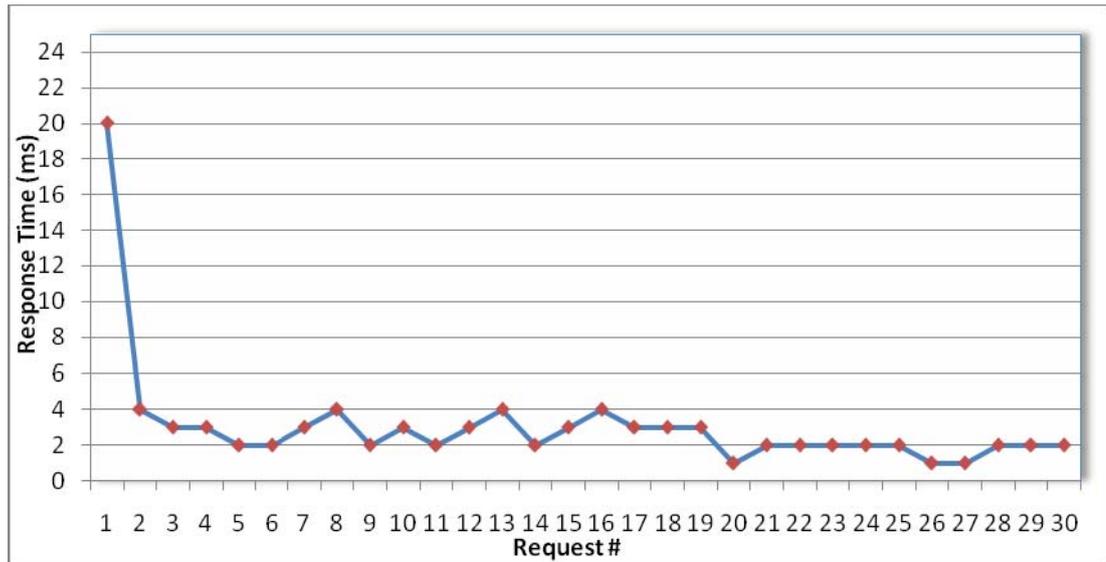


Figure 4.2: Response time and number of requests

Our next experiment has been executed to validate the response time of composed services, following a simple sequence pattern. We use 7 different examples in order to have more precise results. We have 5 different simple Web services and we combine them sequentially in pairs to collect the total response time. This composition of Web services simulates the use of a calculator, while each of these Web services executes a different operation (add, minus, multiplication, division, power).

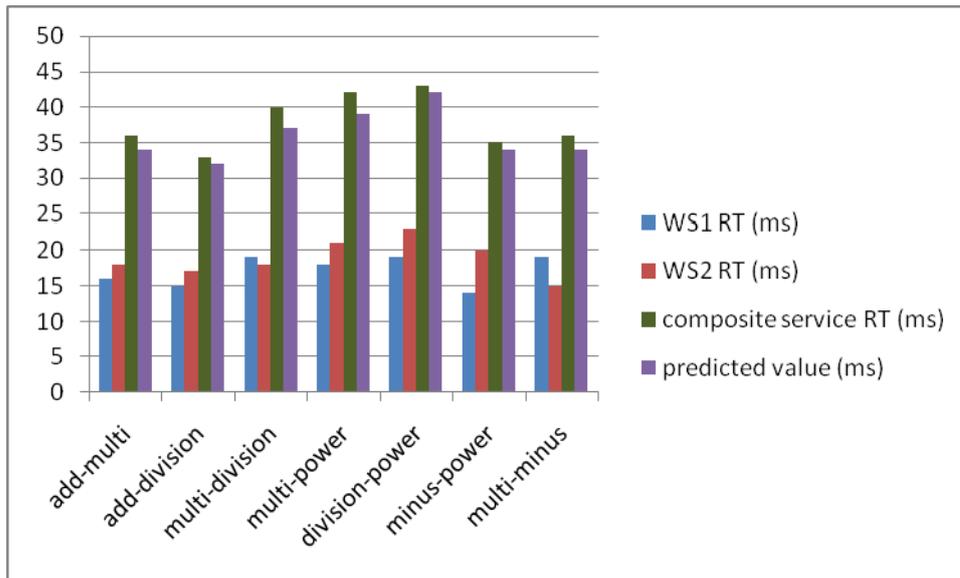


Figure 4.3: Response Time of composed Web services

	WS1 RT (ms)	WS2 RT (ms)	composite service RT (ms)
add-multi	16	18	36
add-division	15	17	33
multi-division	19	18	40
multi-power	18	21	42
division-power	19	23	43
minus-power	14	20	35
multi-minus	19	15	36

Table 4.1: Response time for the constituent and the composed Web service

The experimental results of figure 4.3, confirm the response time formula for complex Web services, proposed in the previous chapter. We can observe that the response time of the composed service with two sequent web services is almost equal to the sum of the response times of the constituent Web services. In all examples, there is deviation of 1-3ms, that is maybe due to the workload of the engine during the initialization for each of the compositions.

A similar experiment (figure 4.4) is used to verify the throughput of composed services, following a sequence pattern.

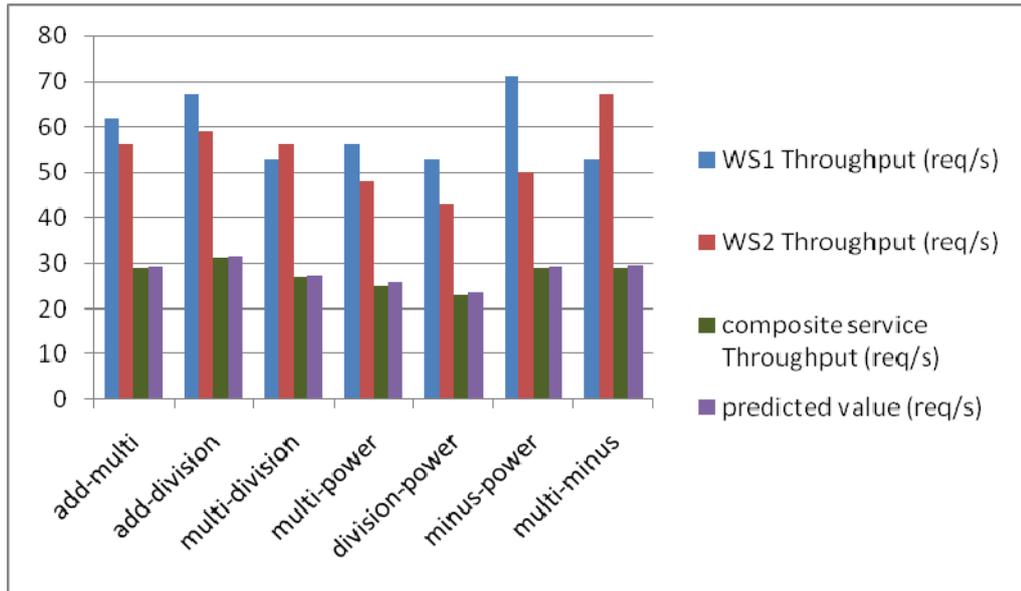


Figure 4.4: Throughput of composed Web services

	WS1 Throughput (req/s)	WS2 Throughput (req/s)	composite service Throughput (req/s)
add-multi	62	56	29
add-division	67	59	30
multi-division	53	56	27
multi-power	56	48	25
division-power	53	43	23
minus-power	71	50	29
multi-minus	53	67	29

Table 4.2: Throughput for the constituent and the composed Web service

The table 4.2 and the corresponding figure illustrates how the throughput of the composed service is affected by the throughputs of the simple services composing it. As we can see, the composed service has less throughput than the throughput of its constituent services and this throughput follows almost precisely the formula we proposed in the previous chapter. Some little variations, as in the previous experiment, are due to the engine's workload for the initializations.

Another experiment we have conducted is to investigate the relationship between the response time and the throughput of a Web service. Response time is the time needed to serve a client's request, while throughput is the number of requests that can be served in unit time. So, we expect that these two metrics are reversely

proportional. We ran 20 different web services on GlassFish, and we monitored their absolute response time and throughput metrics. In Table 4.3, we arrange the executions by ascending order of response time. Figure 4.5, validates exactly the relationship of the two metrics. As response time gets bigger, the throughput gets lesser.

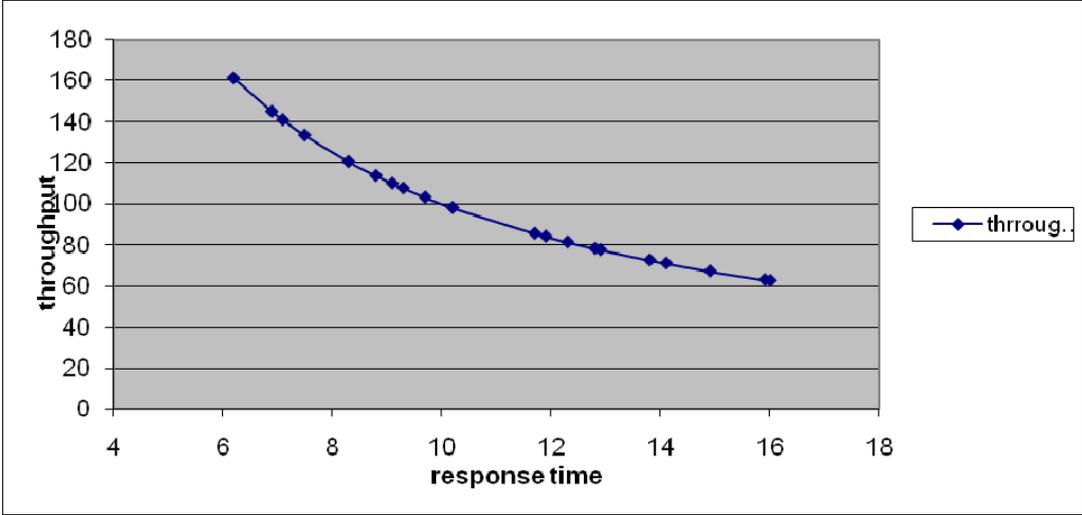


Figure 4.5: Relationship between response time and throughput

Response time	Throughput
6,2	161,29
6,9	144,93
7,1	140,85
7,5	133,33
8,3	120,48
8,8	113,64
9,1	109,89
9,3	107,53
9,7	103,09
10,2	98,04
11,7	85,47
11,9	84,03
12,3	81,30
12,8	78,13
12,9	77,52
13,8	72,46
14,1	70,92
14,9	67,11
15,9	62,89
16	62,50

Table 4.3: Response Time and throughput

In a nutshell, from the previous experiments, we conclude that the composed services “behave” as expected introduced in the previous chapter of this thesis. We must mention here that we have no experiments for reliability and cost metrics, because all our services, used in the examples put 100% reliability while the cost is simply the sum of the costs of the constituent services. Furthermore, we have not carried out any experiment for other composition patterns, since the ActiveBPEL engine we have used for the experimental analysis does not support the extraction of the appropriate values for these patterns.

Chapter 5

5 Related Work

5.1 Monitoring Approaches

This section reviews a number of research and industrial monitoring approaches and discusses their properties in terms of the classification items. A summary of our comparative analysis of all the approaches is presented in Table 5.1.

5.1.1 Monitoring Web service compositions

Barbon, Traverso et. al. [1] propose a solution to the problem of monitoring web service composition, and in particular, to the monitoring of distributed business processes implemented on BPEL for Web services. Their solution has the following main characteristics:

- They devise an architecture where the monitor engine and the BPEL execution engine are executed in parallel on the same application server. This allows for an integration of the two engines, still maintaining the two runtime environments distinct, and keeping the monitors clearly separated from the BPEL processes. As a result, they obtain a clear separation of the business logic from the monitoring task, which allows for an easier adaptation of the business process to the evolving business needs.
- The architecture supports both instance and class monitors: *instance monitors* deal with the execution of a single instance of BPEL business process, while *class monitors* extract information from and/or check the behavior of all the individual instances of a business process. For instance, an instance monitor can check if the bank has rejected the online payment during a specific

session, while a class monitor can provide statistics about on-line payment rejections.

- They provide a novel, rather expressive language for the specification of both instance and class monitors. The language allows for specifying boolean, statistic, and time related properties to be monitored. Beyond monitors of usual boolean properties, they can specify instance monitors that should, e.g., count the number of iterations that are executed in a given session, such as the number of times that a client changes the selected item to buy. They can specify that the monitor should issue an alert if the number of iterations exceeds a given threshold. Moreover, they can specify class monitors that collect information from all existing instance monitors, and check situations of interest — such as the fact that there has been at least one rejection of money transfer by the bank—and/or report on statistics—such as the percentage of payment rejections by the bank.
- Finally, they devise a technique for the automatic generation of the code implementing the instance and class monitors, thus reducing the effort in their design and implementation. Monitors are automatically generated as Java programs that can be deployed in the run-time environment of the monitor engine. To the best of our knowledge, none of the existing approaches for the run-time monitoring of web services supports class monitors and their automated generation from high level specifications.

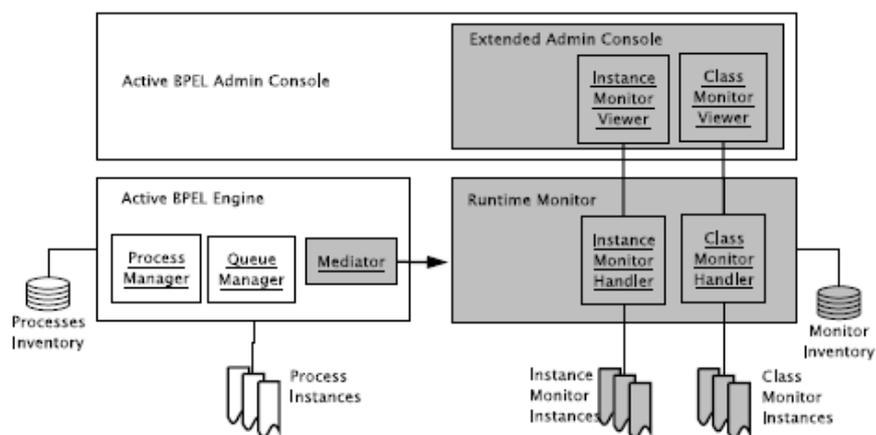


Figure 5.1: The Active BPEL engine extended with the run-time monitor environment

While there have been several works on the automated synthesis of web services and on monitoring web services, much less emphasis have been devoted to the problem of the “assumption-based synthesis and monitoring of web services”, i.e. to the problem of automatically generating composed services by possibly taking into account assumptions at design time, which are then monitored at run-time.

Pistore and Traverso [22] delved into this problem. Given a formal composition requirement, a set of component service descriptions in BPEL and a set of choreographic assumptions expressed in temporal logic, they synthesize automatically an executable BPEL process that, once deployed, satisfies the composition requirement, as well as a set of Java monitors that report at run-time assumption violations. The automated generation of the composed BPEL process takes into account the choreographic assumptions during the synthesis, by discarding behaviors that violate them during the search for a solution. A first advantage of this assumption-based synthesis is that the search for a solution may be simpler and scale up to more complex problems. But, more importantly, this approach is mandatory in the case the composition only exists under the choreographic assumptions. This means that the assumptions are so crucial that, if they are violated, the composition does not make sense. In these cases, assumption-based synthesis and monitoring is the only viable solution.

There are two kinds of monitors: *domain monitors*, which are responsible to check whether the component services respect the protocols, described in their abstract BPEL specification, and *assumption monitors*, which check whether the component services satisfy additional assumptions on their behavior.

Monitors can only observe messages that are exchanged among processes. As a consequence, they cannot know exactly the internal state reached by the evolution of a monitored external service. Non-observable behaviors of a service (such as assign activities occurring in its abstract BPEL) are modeled by τ -transitions, i.e., transitions from state to state that do not have any associated input/output. From the point of view of the monitor, this kind of evolutions of external services cannot be observed, and states involved in such transitions are indistinguishable. Such sets of states are called *belief states*, or simply *beliefs*.

The generation of a domain monitor for an external abstract BPEL process is based on the idea of beliefs and belief evolutions. The domain monitor generation

algorithm (Fig. 5.2) incrementally generates the set MS of beliefs starting from the initial belief ms_0 , by grouping together indistinguishable states of the STS. The beliefs in MS are linked together with (non τ) transitions $MT \subseteq MS \times (I \cup O) \times MS$, as described by function *Evolve*. Beliefs that contain at least one state that is final for the STS are considered possible final states also for the domain monitor, and are stored in MF . Once the algorithm in Fig. 5.2 has been executed, the Java code implementing the domain monitor can be easily generated.

```

procedure build-mon()
   $MS = MT = MF = \emptyset$ 
   $ms_0 = \{\tau\text{-closure}(s_0) : s_0 \in S_0\}$ 
  build-mon-aux( $ms_0$ )

procedure build-mon-aux( $B$ :Belief)
  if  $B \notin MS$  then
     $MS = MS \cup \{B\}$ 
    if  $\exists s \in B. s$  is final then
       $MF = MF \cup \{B\}$ 
    end if
    for all  $m \in (I \cup O)$  do
       $B' = Evolve(B, m)$ 
      if  $B' \neq \emptyset$  then
        build-mon-aux( $B'$ )
         $MT = MT \cup \{< B, m, B' >\}$ 
      end if
    end for
  end if

```

Figure 5.2: The domain monitor generation algorithm

The algorithm for the generation of assumption monitors takes as input the abstract BPEL processes of the external services plus an assumption to be monitored. Assumptions are express in LTL, using as propositional atoms the input/output messages of the component services as well as the properties labeling the states of the STSs modeling these services. To build an assumption monitor, the corresponding LTL formula is mapped onto an STS, which is then emitted as Java code. The evolution of the assumption monitor depends on the input/output messages received by the composite services, which are directly observable by the monitor. However, it also depends on the evolution of the truth values of those basic propositions labeling the states of the components STSs which appear in the LTL formula. These truth values are computed by tracing the evolution of the beliefs of

the component services relevant to the formula. However, it is possible in this case to simplify the “domain” monitor, by pruning out parts of the protocol that are not relevant to tracing the evolution of the basic propositions which appear in the formula. When a stable partition is reached, the reduced monitor is obtained by merging beliefs in the same class of the partition.

5.1.2 Assertion-Based monitoring

Pistore and Traverso [27], specify monitors as assertions that annotate the BPEL code. Assertions can be specified either in the C# language or as pre- post-conditions expressed in the CLIX constraint language. Annotated BPEL processes are then automatically translated to “monitored processes”, i.e. BPEL processes that interleave the business processes with the monitor functionalities.

Their approach provides three main mechanisms to monitor service compositions defined by BPEL programs. These mechanisms correspond to three classes of undesirable behaviors: timeouts, runtime errors and violations of functional contracts. These represent three kinds of service behaviors that can be stated using suitable contracts and may request suitable reaction on the service orchestration side. They express contracts for composed web services using assertions. A non intrusive way of adding assertions is to annotate our BPEL process by inserting them in the form of comments. In this way, the BPEL process remains standard in its definition and executable by any standard BPEL engine. This means that the first step (Fig. 1) in designing a monitored process is to design a standard unmonitored process. After a standard process has been produced, the designer can annotate it with comments representing the contracts he or she wishes to define on timeouts, the desired behavior of the process in presence of errors, and functional contracts in terms of pre- and post-conditions. These comments represent the core of the design overhead necessary for monitoring processes. In fact, these comments are automatically translated into the sequence of BPEL activities that augment the original process in order to make it monitored.

Contracts [3] are translated in different ways: Timeouts and errors in service implementations can be handled using standard BPEL and good design patterns

functional contracts (pre- and post-conditions) require dedicated monitors. As we mentioned, functional contracts are monitored by special-purpose components called monitors. The monitor is itself a web service and consequently becomes a part of the monitored composition. The monitored process reacts to misbehaviors by terminating the execution and signaling the detected problem.

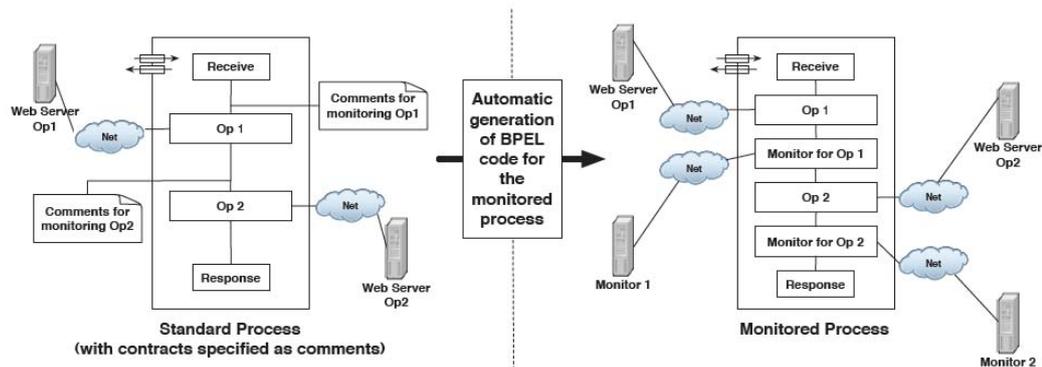


Figure 5.3: A standard process annotated with contracts transformed into a monitored

Baresi and Guinea [2], also extended their work with the ability to perform “dynamic monitoring”, i.e. the ability to specify monitoring rules that are dynamically selected at run-time, thus providing a capability to dynamically activate/deactivate monitors, as well as to dynamically set the degree of monitoring at runtime. Monitoring rules abstract web services into UML classes that are used to specify constraints on the execution of BPEL processes. Assertions, are specified in WS-COL (Web Service Constraint Language), a special purpose language that extends JML (Java Modeling Language), with constructs to gather data from external sources. Monitoring rules are defined with parameters that specify the degree of monitoring that has to be performed at run-time. The user can instantiate dynamically these parameters at run-time, changing in this way the amount of monitoring that is performed.

5.1.3 Run-Time monitoring

Requirement-Based monitoring

Run-time requirements monitoring has been the focus of different strands of requirements engineering research which have investigated: (i) ways of specifying requirements for monitoring and transforming them into events that can be monitored at run-time; (ii) the development of event-monitoring mechanisms; (iii) the development of mechanisms for generating system events that can be used in monitoring; and (iv) the development of mechanisms for adapting systems in order to deal with deviations from requirements at run-time.

The need for run-time requirements verification is very important, especially for service-based software (SBS) systems (i.e. systems which are composed from autonomous web services co-ordinated by some composition process). This is because the web-services that constitute an SBS system may not be specified at a level of completeness that would allow the application of static verification methods, and some of these services may change dynamically at run-time causing unpredictable interactions with other services.

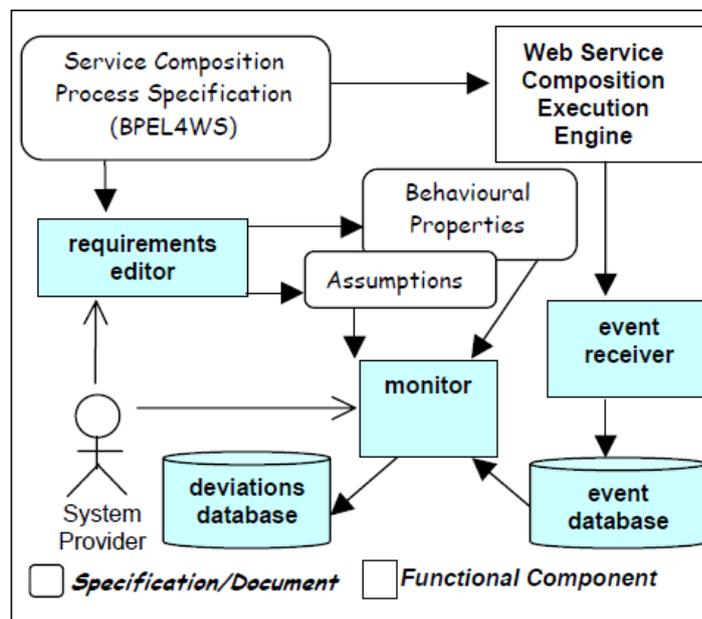


Figure 5.4: Monitoring framework

Mahbub and Spanoudakis [22] propose a framework that supports the run-time monitoring of behavioral properties of an SBS system or assumptions about the behavior of the different web services that constitute it or agents in its environment. Behavioral properties are automatically extracted from the specification of the composition process of the SBS system. Assumptions are additional requirements about the behavior of agents interacting with the system, or the individual services of it. Assumptions are specified in event calculus using an XML schema they have developed to support the representation of event calculus formulas.

Both behavioral properties and assumptions are expressed in *event calculus* and are being monitored by using a variant of techniques developed for checking integrity constraints in temporal deductive databases. The choice of event calculus as the requirements representation language has been motivated by the need to express the properties to be monitored in a formal language with well-defined semantics that allows: (a) the specification of temporal constraints and (b) reasoning based on the inference rules of first-order logic.

This framework can monitor three different types of deviations from behavioral properties and assumptions. These are: (i) violations of assumptions by the recorded system behavior, (ii) violations of behavioral properties and assumptions by the expected system behavior (i.e. the behavior that would have been exhibited by the system if assumptions other than the one being checked had been satisfied), and (iii) cases of unjustified system behavior that may arise when a system acts incorrectly due to incorrect information about its state.

Monitoring is performed in parallel with the normal operation of an SBS system without interrupting it. This is possible by intercepting events which are exchanged between the composition process of an SBS system and its services and the effects of these events on the state of the composition process of the system. This approach makes run-time monitoring non intrusive as: (a) it does not affect the performance of SBS systems, and (b) it does not require the instrumentation of the code of the composition process of SBS systems or their services to generate the events which are required for monitoring.

Event-based monitoring

Given a monitoring-enabled infrastructure to detect and route service operational events, it is imperative that these events be processed efficiently, and that the QoS metric values be computed and saved efficiently as well. Although most complex event processing systems support high throughput of events, they primarily focus on event filtering and compound event detection. They do not address metric computation, where event data triggers and contributes to a complex flow of computation. Further, they don't consider the issue of state persistence. Zeng et al. [34] advocate a series of model analysis techniques to improve event throughput in a monitoring environment.

Event-driven rule-based programming is user friendly, particularly for business integration developers. However, because of the overhead in locating rules to be executed at runtime, the event-driven model does not lend itself to efficient execution, especially when the number of rules is very large, such as in the case of service QoS monitoring. In this design, the rule-based model is transformed to a state-based model, wherein statecharts are adopted to reorganize the rules. The rationale for such a model transformation is that statecharts organize the rules by states, which can greatly reduce the overhead in locating rules at runtime.

The construction of statecharts is based on user-defined ECA rules: a state represents either an event or a metric, while a transition between two states represents the triggering relationship (see figure 5.5). For example, if the event pattern is a service operational event in an ECA rule, then there is a transition from the event state to the metric state. In another case, the event pattern is the value change of a metric, and the corresponding transition is from one metric state to another metric state.

ECA rule type	Control-flow Transitions in Statechart
Event pattern is a service operational event	 A diagram showing a rounded rectangular box labeled "Event" on the left, connected by a horizontal arrow to a rounded rectangular box labeled "Metric" on the right.
Event pattern is value change of a metric	 A diagram showing a rounded rectangular box labeled "Metric1" on the left, connected by a horizontal arrow to a rounded rectangular box labeled "Metric2" on the right.

Figure 5.5: Transforming the ECA rules to Statecharts

Zeng et al. [34] propose a hybrid approach, in which, state transition logic is interpreted, while the expression in a rule is compiled into standalone executable code. The advantages of such a hybrid approach are twofold. On the one hand, by interpreting the state transition logic, the computation engine can plan the execution of rules in finer granularity. On the other hand, the execution of an individual expression is done by executing pre-compiled code, which enjoys the efficiency of the compilation approach.

Run-time monitoring using WS-Policy

WS-Policy is emerging as the standard way to describe the properties that characterize a Web service. By means of this specification, the functional description of a service can be tied to a set of assertions that describe how the Web service should work in terms of aspects like security, transactionality, and reliable messaging. These assertions can be used to express both functional and non-functional aspects. Policies can be defined by several actors and during different phases of the Web service life-cycle. Besides implementing the application, service developers also specify the properties that must hold during the execution regardless of the platform on which the services will be deployed (service policies). On the other hand, service providers specify the features supported by the application servers on which services are deployed (server policies). The intersection of service and server policies results in *supported policies*, which define the properties of the services deployed on a specific platform. Finally, Web service users state the features that should be supported by the services they want to invoke (requested policies). By combining requested policies and supported policies, we obtain the so called *effective policies*.

Effective policies represent the set of assertions that specify the properties of a Web service deployed on a particular server and invoked by a specific user. The Web service to which effective policies apply is linked by definition and it can be a simple Web service or a WS-BPEL process. Once effective policies are derived, services should be monitored at runtime to guarantee that they offer the service levels stated by their associated policies.

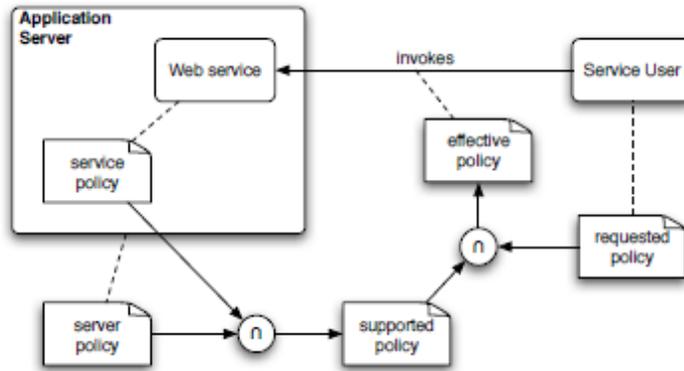


Figure 5.6: WS-Policy definitions and attachments

The tradeoff between monitoring and performance might be influenced by many different factors. We cannot define a strict relationship between WS-BPEL processes and monitoring directives. Users must be free to change them to cope with new and different needs. For example, the execution of these processes in different contexts might require a heavier burden in terms of monitoring, while when selected services are well-known and reliable, users might decide to privilege performance and adopt a looser monitoring framework.

These considerations led Baresi, Guinea and Plebani [2] to propose monitoring directives as stand-alone (external) *monitoring policies* rendered in WS-Policy. These constraints do not belong to the workflow description, that is, the WS-BPEL process, but they are weaved with it at deployment-time. For each policy, the embedded location indicates the point of the process in which BPEL substitutes the WS-BPEL invoke activity with a call to the monitor manager, which is then in charge of evaluating the policy and call the service if it is the case. BPEL also adds an initial call to the monitoring manager, to send the initial configuration (such as the priority at which the process is being run) to initialize it, and a final call to communicate it has finished executing the business logic and that resources can be released. BPEL only adds calls to the monitoring manager. This means that policies can change without re-instrumenting the process.

5.1.4 Planning and monitoring service requests

A significantly different approach is proposed by Lazovik et al. [21]. They present a planning architecture (with a specially tailored run-time environment) in which service requests are presented in a high-level language called XSRL (Xml Service Request Language). They adopt a proprietary orchestrated approach to collaboration, since they claim that current standards, like BPEL, do not have the necessary flexibility to satisfy user requirements that heavily depend on run-time context information.

The planning architecture is based on a continuous interleaving of planning steps and execution steps. Because BPEL lacks formal semantics, the authors decided to extrapolate state-transition systems from BPEL specifications and to enrich them with domain operators and constructs.

This framework is based on reactive monitoring. In particular, designers can define three kinds of properties: (1) Goals that must be true before transitioning to the next state (2) goals that must be true for the entire process execution, and (3) goals that must be true for the process execution and evolution sequence. The XSRL language also allows for the definition of constraints as boolean combinations of linear inequalities and boolean propositions. It provides sequencing operators such as “achieve-all”, “before” and “then”, “prefer” goal x “to” goal y, and “then”. It also defines a number of operators that can be used on the propositions themselves, defining how these propositions should be satisfied such as “vital” and “optional”. The delivery platform continuously loops between execution and planning. In particular, the latter activity is achieved by taking into account context and the properties specified for the state-transition system. This makes it possible to discover, each time it is undertaken, whether a property has been violated by the previously executed step, or if execution is proceeding correctly.

5.1.5 Monitoring tools

Cremona

Cremona [13] is a proposal from IBM, which stands for “Creation and Monitoring of WS-Agreements”, is a special-purpose library devised to help clients and providers in the negotiations and life-cycle management of WS-Agreements (i.e., their creation, termination, run-time monitoring, and re-negotiation).

WS-Agreement [26] specifies an XML-based language for creating contracts, agreements, and guarantees from offers between a service provider and a client. An agreement may involve multiple services and includes information on the agreement parties, references to prior agreements, service definitions, and guarantee terms. In an agreement the service definition is part of the terms of the agreement and must be established prior to the creation of the agreement. The motivations for the design of WS-Agreement stem out of QoS concerns, especially in the context of load balancing heavy loads on a network of Web services.

The Cremona framework provides an “Agreement Provider” component, whose structure incorporates, among other things, a “Status Monitor”. This component is specific to the system providing the service. By consulting the resources available on the system and the terms of an agreement, it helps decide whether a negotiation proposal should be accepted or refused. Once an agreement has been accepted by both parties (the client and the provider), its validity is checked at run-time by a “Compliance Monitor”, a sophisticated system-specific component that can check for violations as they occur, predict violations that still have to occur, and take corrective actions. Since both monitoring components are system dependent, designers are guaranteed great flexibility in terms of the properties they can check.

Colombo

Colombo [13] is a lightweight platform for developing, deploying, and executing service-oriented applications. It provides optimized, native runtime support for the service-oriented-computing model, as opposed to the approach of layering service-oriented applications on a legacy runtime. This approach allows

Colombo to provide high runtime performance, a small footprint, and simplified application development and deployment models. The Colombo runtime natively supports the full Web Services (WS) stack, providing transactional, reliable, and secure interactions among services. It defines a multi-language service programming model that supports, among others, Java™ and Business Process Execution Language for Web Services (BPEL4WS) service composition, and offers a deployment and discovery model fully based on declarative service descriptions (Web Service Description Language [WSDL] and WS-Policy).

Colombo manages incoming and outgoing messages by passing them through two corresponding pipes of dedicated policy verifiers and enforcers (i.e. one for each kind of policy supported by the system), it can discover erroneous behavior in a timely fashion, but is intrusive in nature. It provides support for important issues, such as security.

GlassFish

As mentioned in the previous chapter, GlassFish was used in our work. It is an open-source community implementation of a server for Java EE 5 applications. Regarding the monitoring of deployed services, GlassFish provides a number of specific tools. The nature of the monitored aspects depends on the level of monitoring chosen for a given service. There are three possible levels: *low*, which monitors response times, throughput, and the total number of requests and faults; *medium*, which adds message tracing under the form of content visualization; and *off*, in which no data is collected. Captured information can also be automatically aggregated to obtain “minimum response times”, “maximum response times”, “average response times”, etc.

IBM Tivoli Composite Application Manager for SOAs

Another similar approach is the IBM Tivoli Composite Application Manager [15] for SOAs, which is an application manager that uses an event-based collaboration paradigm, implemented through a special-purpose integration bus.

In this environment, traditional tools that monitor individual resources performance typically cannot solve composite application performance and

availability problems. Instead, Web services need to be incorporated into the end-to-end management domain over composite applications and resources that support an SOA environment. Because many Web services are used to make mainframe applications and middleware available at the front end, it is not adequate to monitor and manage only at the Web services level. Instead, businesses need to view Web services as part of their end-to-end infrastructures. Otherwise, operations and development teams waste countless hours trying to identify, isolate and fix problems — all while poorly performing composite applications negatively affect the top- and bottom-line results of the business.

IBM Tivoli Composite Application Manager (ITCAM) for SOA delivers unparalleled, integrated management tools for Web and enterprise infrastructure that help maintain availability and performance on demand business. With ITCAM for SOA, you can monitor, manage and control the service layer of your IT architecture. This application manager uses an event-based collaboration paradigm, implemented through a special-purpose integration bus. Messages enter and leave the bus continuously, passing through special components called the “ServiceBusInbound” and the “ServiceBusOutbound”, making it easy to monitor their behavior and, in particular, their performance. However, the application manager lacks the specially tailored tools present in other similar approaches.

BP-Mon: Query-Based Monitoring of BPEL Business Processes

The BP-Mon [4] system is part of BP-Suite, a novel tool suite based on the BPEL standard. BP-Suite offers a uniform, query-based, user-friendly interface that gracefully combines the analysis of process specifications, monitoring of run time behavior, and log analysis, for a comprehensive process management. BP-Suite consists of three tightly coupled query subsystems: BP-QL allows one to query and analyze BP specifications; BP-Mon allows for monitoring the execution of BP instances; and BP-Ex allows for a posteriori analysis of their execution traces (logs).

BP-Mon allows users to monitor process instances at run-time and to visually define monitoring tasks and associated reports, using a simple intuitive interface similar to those used for designing BPEL processes. Also, BP-Mon queries are

translated to BPEL processes that run on the same execution engine as the monitored processes.

BP-Mon system makes the following contributions:

- *Query language.* The system is based on an intuitive graphical query language that allows for simple description of the execution patterns to be monitored. The BP-Mon query language is an adaptation of the sister query languages used in BP-QL (for specification analysis) and BP-Ex (for logs analysis), to run-time monitoring. In all three query languages, the data (i.e. the BP specification or its execution traces) is abstractly viewed as a nested set of Directed Acyclic Graphs. A query consists of two parts. The first specifies the execution patterns that are of interest to the user. The second part of the query, consists of *Report icons* that can be attached to the patterns.

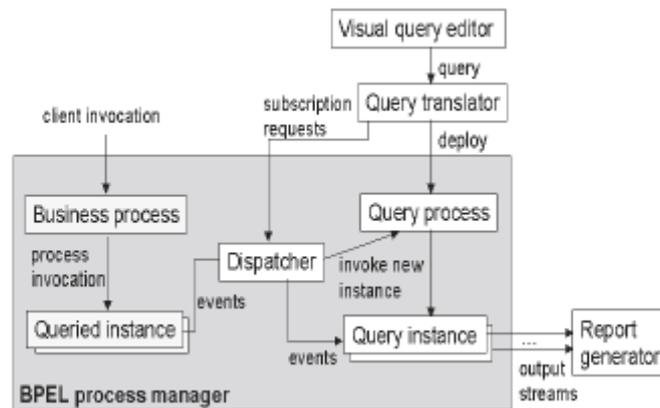


Figure 5.7: BP-Mon architecture

- *Deployment.* To support flexible deployment, the system compiles a BP-Mon query q into a BPEL process specification S , whose instances perform the monitoring task. As for all standard BPEL specifications, S can now be automatically compiled into executable code to be run on the same BPEL application server as the monitored BP.

In summary, BP-Mon allows one to design complex monitoring tasks that deal with both events and flow; it offers easy, user friendly design of such tasks; and it compiles these tasks into standard BPEL processes, thus providing easy deployment, portability, and minimal overhead.

5.2 Comparing monitoring approaches

As described below, there are many monitoring approaches. Each of them has its advantages and disadvantages. In this section we make a comparison of the previous approaches and in the end there is a pivot table of the comparative analysis.

On the one hand, the approach described in [4,5] provides some advantages with regard to [22]. First, monitors are themselves services implemented in BPEL. As a consequence, they can run on standard BPEL engines without requiring any modification. Second, annotations of BPEL processes with assertions constitute an easy and intuitive way to specify monitor tasks. Finally, the approach is extended to dynamic monitoring, a feature that is not provided in our framework.

On the other hand, in [22] Pistore and Traverso's approach allows for the monitoring of properties that depend on the whole history of the execution path. These kinds of monitors would be hard to express as assertions. Moreover, they allow for a clearer separation of the business logic from the monitoring task than in [4,5], since they generate an executable monitor that is fully distinguished from the executable BPEL that runs the business logic. Finally, their monitors can capture misbehaviors generated by the internal mechanisms of the BPEL execution engine. For instance, since there is no way to guarantee that a message is sent to a process instance only when the instance is ready to consume it, in BPEL, messages can be consumed in a different order from how they are received: indeed a process may receive a message that it is not able to accept at the moment, which can be followed by another message that can instead be consumed. The first message can be consumed later on by the process, or may never be consumed. This phenomenon, called *message overpass*, cannot be captured by monitors based on assertions that annotate the BPEL code.

There are also some similarities between Mahbub and Spanoudaki's approach in [22] and Pistore and Taverso's approach. Both share the idea to have a monitor that is clearly separated from the BPEL processes. Another similarity is that the framework allows for specifying requirements that represent either behavioral properties or assumptions to be monitored. [13]

Approach Name	Type of Properties	Collaboration paradigm	Collecting monitoring data	Timeliness	Abstraction level	Validation technique	Monitoring goals
Dynamo	Mainly functional (and simple non-functional properties)	BPEL-based orchestrations	Collected by the process itself, or through external data sources	Blocking pre- and post-conditions	Programming level	Assertion-checking	Tools for composition providers who need to monitor the external services used
Requirements monitoring	Mainly functional (and simple non-functional properties)	BPEL-based orchestrations	An interceptor component listens for low-level engine events	Post-mortem	Low-level sequences of engine events	Variant of integrity checking in temporal deductive databases	Tools for composition providers who need to monitor the external services
Planning and monitoring Service Requests	Process and evolution sequence goals	Proprietary orchestratuin-based delivery framework	Collected within the propriataty framework	Errors discovered as soon as they occur	Requirements and specification level (market domain terminology)	Assertion-checking approach	Tools for composition providers who need to monitor process evolution
Cremona	Functional and non-functional and properties of histories of interactions	No specific paradigm, but any interaction between a caller and the provider	Server-side regarding the interaction channel and the system's resources	Reactive approach	WS-Agreement templates with different property description languages	Implementation-specific techniques	Tools for service providers who need to monitor agreements with their clients
Colombo	Mainly non-functional properties (WS-policy)	Optimized middleware for SOA that supports BPEL	Through a pipe of dedicated policy-specific verifiers	Before a message leaves the system, or before the incoming message is processed	Service, operation, or message level	Validation is policy dependent	Tools for service providerts who need to monitor policy compliance of incoming and outgoing messages
GlassFish	Mainly non-functional properties	Proprietary deployment infrastructure	Response times, throughput, number of requests and message tracing	No automatic analysis. Timeliness does not depend on the system	Three standard macro-degress or monitoring	No automatic validation	Tools for service providers who need to monitor statistics of client-service interactions
IBM Tivoli Composite Application Manager	Mainly non-functional properties	Event-based system (integration bus)	Messages as they enter or leave the integration bus	No automatic analysis. Timeliness does not	WS-Policy for QoS	No automatic validation	Tools for service providers who can pesimalize

				depend on the system			monitoring on-top of the service bus structure
BP-Mon	Mainly non-functional properties	Event-based system	Collected by the process itself, or through external data sources	No automatic analysis. Timeliness does not depend on the system	Programming level	No automatic validation	Tools for service providers who want to monitor BPs at run-time and analyze the logs

Table 5.1: Comparing monitoring approaches

5.3 Computing metrics of composed services

Menasce [24] proposes a method to estimate the throughput of a composed service from those of its constituent Web services. To make his work clearer, he uses Web service flow graphs (WSFG), whose nodes are either Web Sites or Web services. A directed edge between nodes a and b indicates that a uses the service b . The label on the edge (a,b) , called the *relative visit ratio*, is the average number of times node b is visited per visit to node a . So on average, each travel booking request to the travel site generates V_a requests to the airline Web service, V_h requests to the hotel Web service, and V_c requests to the car rental Web service.

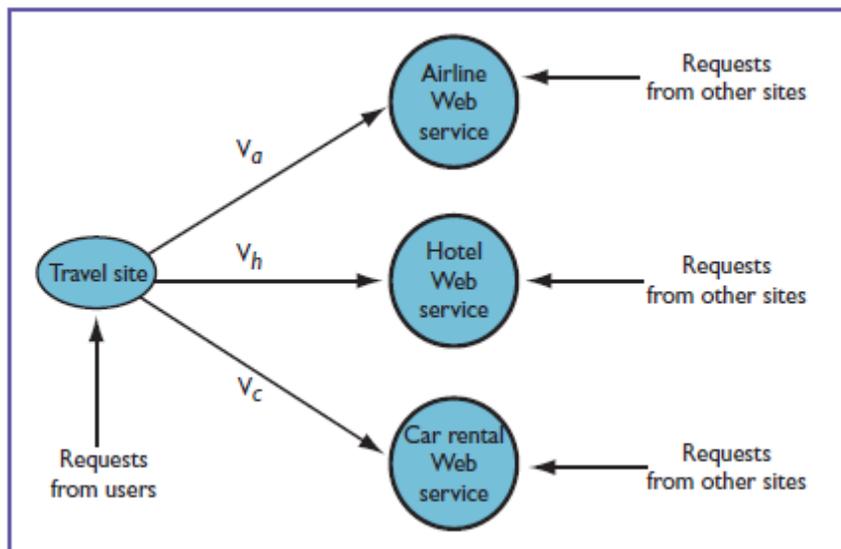


Figure 5.8: The travel site Web service

The author's goal is to establish an upper bound on the throughput XTA of the travel site based on the throughputs of the three Web services it uses. The throughput

X_a of the airline service therefore needs to be at least equal to $V_a \times X_{TA}$, since that service must be able to serve all requests it receives from the travel site as well as all requests coming from other sites that use its service. Thus, we have the equations:

$$X_a \geq V_a \times X_{TA} \quad (1)$$

$$X_h \geq V_h \times X_{TA} \quad (2)$$

$$X_c \geq V_c \times X_{TA} \quad (3)$$

where X_a , X_h , and X_c represent the throughputs of the airline, hotel, and car rental Web services, respectively.

A combination of 1-3 is used to establish an upper bound on the throughput of the travel site:

$$X_{TA} \leq \min \left\{ \frac{X_a}{V_a}, \frac{X_b}{V_b}, \frac{X_c}{V_c} \right\}$$

To see the usefulness of this equation, suppose that the throughput of the airline, hotel, and car rental Web services is 20 requests/sec, 15 requests/sec, and 10 requests/sec, respectively, and that on average, each travel site request will visit the airline Web service four times, the hotel Web service twice, and the car rental service only once. So, using the previous equation:

$$X_{TA} \leq \min \left\{ \frac{20}{4}, \frac{15}{2}, \frac{10}{1} \right\} = 5 \text{ requests/sec.}$$

This equation says that in order for the travel site to increase the upper bound on its throughput, it would need to use a better airline Web service, because this is the Web service that limits the maximum throughput of the travel site. Alternatively, the travel site could try to reduce the number of times it has to invoke the airline Web service per transaction.

Hwang et al. [27] propose a probabilistic QoS model and computation framework for Web Services-Based workflows. They identify a set of QoS metrics in the context of Web services. Each QoS measure of a Web service is regarded as discrete random variable with a probability mass function (PMF).

They also explore alternative algorithms for computing probability distribution functions of WS-workflow QoS. The efficiency and accuracy of these algorithms are compared.

```

ComputeQoS(A: a WS-workflow activity)
{
  IF A.type ≠ ATOMIC THEN {
    FOR (each activity  $t \in A.activities$ ) DO
      ComputeQoS( $t$ );
    IF A.construct = SEQUENTIAL THEN
      A.QoS = SequentialQoS(A.activities);
    ELSEIF A.construct = PARALLEL THEN
      A.QoS = ParallelQoS(A.activities);
    ELSEIF A.construct = CONDITIONAL THEN
      A.QoS = ConditionalQoS(A.activities);
    ELSEIF A.construct = FAULT_TOLERANT THEN
      A.QoS = FaultTolerantQoS(A.activities);
    ELSE // A.construct = LOOP
      A.QoS = LoopQoS(A.activities);
  }
  ELSE Estimate the QoSs of A and put them in A.QoS;
};

```

Figure 5.9: Pseudo code for computing QoS of a WS-workflow

SequentialQoS(A.activities), *ParallelQoS*(A.activities), *ConditionalQoS*(A.activities), *FaultTolerantQoS*(A.activities), *LoopQoS*(A.activities) are used to compute cost, response time, reliability and fidelity QoS metric values for sequential, parallel, conditional, fault tolerant, and loop constructs respectively.

5.4 Comparison of our work with existing studies

In this section, we make a comparison of the afore-mentioned related work with ours. As already analyzed, the existing work focus basically on requirement-based and event-based systems. The first ones check if the pre-conditions that were defined are fulfilled after the execution, while the event-based systems are waiting for specific events to get a monitored value. Both approaches are separated from BPEL processes. As far as the computation of QoS for composed Web services is concerned, the existing work is limited to the very basic metrics for compositions following the sequence pattern.

Our approach now, combines a monitoring system with SLA compliance. In essence, it acts as a mediator between a monitoring system and the SLA document. If we would like to classify our work to a general monitoring group, we could say that

it is requirement-based. The requirements are expressed in the WSLA document and after the monitoring process, they are checked for their correctness. In the computation of QoS for composed Web services area, we have extended the current state-of-the-art work to express formulas for more metrics and for many more composition patterns.

Chapter 6

6 Conclusion and Future Work

Our work is separated in two different, but related knowledge domains of Web services: monitoring the QoS compliance of Web services using SLAs and computing the QoS of composed Web services.

As far as the QoS compliance is concerned, we propose a system that combines the monitoring system of Sun Application Server GlassFish with SLA documents, written in WSLA, an XML-based language. Our system requires to have available, both the Web services to be deployed on the application server and the corresponding SLA of the Web service. If these requirements are fulfilled, then the system compares the pre-agreed metric conditions with the monitored metric values and if there is a violation, it alerts the interested parties with the detected differences. The results are satisfying, compared to other similar systems, as they seem to be fast and precise. Furthermore, these results are very helpful for the service provider, who can take corrective actions in case metric values don't comply with SLAs. In addition, statistical analysis of the results can extract many useful conclusions to assist the provider in developing new and more effective Web services.

As far as the computing of QoS is concerned, we propose some basic metric formulas, provided that we know the composition pattern, which may be a combination of other patterns. Based on this, we use the appropriate formulas to compute the response time, the throughput, the reliability and the cost of the composed Web service. From our experimental results, we conclude that the proposed types are precise enough and can be used in every complex service compositions, described with the afore-mentioned patterns. Our experiments focus basically on using a variety of composition patterns in the BPEL engine and the results validate to a great extent our work for computing metrics for composed services.

Finally, we believe that the main issue remaining for future work is to examine more formulas for all possible metrics and composition patterns. Also, it would be ideal to make our monitoring system work at runtime with the Web service execution engine. In this way, the provider could take statistical results and correct any violations very quickly and without any mediators. Another interesting direction is to focus on the corrective actions after a violation is detected. This can be performed by another management service that would take as input the condition evaluation results and then, if appropriate, would try to make the service compliant with the SLA document.

Bibliography

- [1] **F. Barbon, P. Traverso, M. Pistore, M. Trainotti.** "Run-Time Monitoring of Instances and Classes of Web Service Compositions". In *Procs of ICWS'06*, pages 63-71, Salt Lake City, Utah, USA, July 2006.
- [2] **L. Baresi, S. Guinea and P. Plebani.** "WS-Policy for Service Monitoring". In *Procs of TES 2005*, pages 72-83, Trondheim, Norway, September 2006.
- [3] **L. Baresi, C. Ghezzi and S. Guinea.** "Smart Monitors for Composed Services". In *Procs of ICSOC'04*, pages 308 - 315, New York, USA, November 2004.
- [4] **C. Beerli, A. Eyal, T. Milo and A. Pilberg.** "BP-Mon: Query-Based Monitoring of BPEL Business Processes". In *SIGMOD Record*, Vol. 37, No 1, March 2008.
- [5] **P. Bianco Philip, G. Lewis and P. Merson.** "Service Level Agreements in Service-Oriented Architecture Environments". *Technical Note of Software Engineering Institute*, September 2008.
- [6] **D. Bianculli and C. Ghezzi.** "Monitoring Conversational Web Services". In *Procs of IW-SOSWE*, pages 15-21, Dubrovnik, Croatia, September 2007.
- [7] **A. Bucchiarone and S. Gnesi.** "A Survey on Services Composition Languages and Models". In *Procs of WS-MaTe 2006*, Palermo, Italy, June 2006.
- [8] **A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer and A. Youssef.** "Web services on demand: WSLA-driven automated management". In *IBM Systems journal*, Vol. 43, No 1, 2004.
- [9] **A. Dan, H. Ludwig and G. Pacifici.** "Web Services Differentiation with Service Level Agreements". In *Tech Report of IBM Corp.*, 2003.
- [10] **A. Daniel, A. Barbir, C. Ferris, S. Garg.** "*Web Services Architecture Requirements*", February 2004. <http://www.w3.org/TR/wsa-reqs/#id2604831>.
- [11] **A. David and F. Xavier.** "SALMon, Service Level Agreement Monitor". In *Procs of 7th International Conference on Composition-Based Software Systems*, pages 224-227, Madrid, Spain, February 2008.

- [12] **M. Debusmann and A. Keller.** "SLA-driven management of distributed systems using the Common Information Model". In *Procs of IM 2003*, pages 563-576, Colorado, USA, March 2003.
- [13] **C. Ghezzi and S. Guinea.** "Run-Time Monitoring in Service-Oriented Architectures". In *Test and Analysis of Web Services*, Springer Publications, pages 237-264, September 2007.
- [14] **S. Guinea.** "Self-healing Web Service Compositions". In *Procs of ICSE '05*, pages 655-658, St. Louis MO, USA, May 2005.
- [15] **IBM.** "IBM Tivoli Composite Application Manager for SOAs", In *Technical Report of IBM*, 2006.
- [16] **R. Jurca, W. Binder and B. Faltings.** "Reliable QoS Monitoring Based on Client Feedback". In *Procs of WWW'07*, pages 1003-1012, Banff, Alberta, Canada, May 2007.
- [17] **R. Kassab and Aad van Moorsel.** "Mapping WSLA on Reward Constructs in Mobius", In *Procs of UKPEW 2008, London, England, July 2008*.
- [18] **A. Keller and H. Ludwig.** "The WSLA Framework: Specifying and Monitoring of Service Level Agreements for Web Services". *IBM research report RC22456*, May 2002.
- [19] **A. Keller and H. Ludwig.** "Defining and Monitoring Service Level Agreements for dynamic e-Business". In *Procs of LISA 2002*, pages 189-204, Philadelphia, PA, USA, November 2002.
- [20] **D. Lamanna, J. Skene and W. Emmerich.** "SLAng: A language for defining Service Level Agreements". In *Procs of FTDCS'03*, pages 100-106, San Juan, Puerto Rico, May 2003.
- [21] **A. Lazovik, M. Aiello and M. Papazoglou.** "Planning and monitoring the execution of web service requests". *International Journal on Digital Libraries*, pages 235-246, Vol. 6, No 3, June 2006.
- [22] **K. Mahbub and G. Spanoudakis.** "A Framework for Requirements Monitoring of Service Based Systems". In *Procs of ASE'04*, pages 379-384, Linz, Austria, September 2004.
- [23] **K. Mahbub and G. Spanoudakis.** "Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementaion and Evaluation Experience". In *Procs of ICWS 2005*, pages 257-265, Orlando, Florida, USA, June 2005.

- [24] **D. Menasce.** "QoS Issues in Web Services". *IEEE Internet Computing*, pages 72-75, vol. 6, no. 6, November-December 2002.
- [25] **M. Musicante and E. Potrich.** "Expressing Workflow Patterns for Web Services: The Case of PEWS". *Journal of Universal Computer Science*, pages 903-921, Vol. 12, No 7, 2006.
- [26] **M. Papazoglou.** "Web Services: principles and technology", Pearson Publications, 2008.
- [27] **M. Pistore and P. Traverso.** "Assumption-Based Composition and Monitoring of Web Services", In *Test and Analysis of Web Services*, Springer Publications, pages 307-335, September 2007.
- [28] **F. Rosenberg, C. Platzer and S. Dustdar.** "Bootstrapping Performance and Dependability Attributes of Web Services". In *Procs of ICWS'06*, pages 205-212, Chicago, USA, September 2006.
- [29] **H. San-Yih, Wang H., S. Jaideep and P. Raymond.** "A Probabilistic QoS Model and Computation Framework for Web Services-Based Workflows". In *Procs of ER2004*, pages 596-609, Sanghai, China, November 2004.
- [30] **A. ShaikhAli, F. Rana, R. Al-Ali and W. Walker.** "UDDIe: An Extended Registry for Web Services". In *Procs of SAINT'03 Workshops*, pages 85-89, Orlando, Florida, January 2003.
- [31] **V. Tosic V, B. Pagurek, K. Patel, B. Esfandiari and W. Ma.** "Management Applications of the Web Service Offerings Language (WSOL)". In *Procs of CAiSE'03*, pages 564-586, Klagenfurt/Velden, Austria, June 2003.
- [32] **U. Wahli, O. Burroughs, O. Cline, A. Go and L. Tung.** "Web Services Handbook for WebSphere Application Server Version 6.1", *IBM Red Books Publication*, 2006.
- [33] **E. Wustenhoff.** "Service Level Agreement in the Data Center". *Sun BluePrints*, April 2002.
- [34] **L. Zeng, H. Lei and H. Chang.** "Monitoring the QoS for Web Services", In *Procs of ICSOC 2007*, Springer Publications, pages 132-144, Vienna, Austria, September 2007.
- [35] **C. Zhou, L. Chia, S. Bilhanan and B. Lee.** "UX – An Architecture Providing QoS-Aware and Federated Support for UDDI". In *Procs of ICWS'03*, pages 171-176, Las Vegas, USA, June 2003.