Runtime Monitoring of Service Based Systems

KHALED MAHBUB

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

at

City University, London

Department of Computing

November 2006

Table of Contents

List of Tables

List of Figures

Acknowledgements

First, I extend my deepest gratitude to my supervisor, Professor George Spanoudakis, for his constant encouragement and assistance. He is a patient and nurturing mentor, who could always boost my confidence in times of self-doubt or frustration. His insistence on perfection was relentless and sometimes painful, but undoubtedly this will serve as a guide for the rest of my life. Without his guidance and continuous cooperation throughout this research it would not have been possible to finish this thesis. I am honoured to have his name on this work.

I wish to express my sincere thanks to my co-supervisor Dr. Andrea Zisman for her valuable advices during this research.

I like to thank the School of Informatics for the financial support for this research. I also like to remember every member of the Technical Support Team (TST), in the School of Informatics for their tireless services. Without their assistance this journey could have been dreadful to me. They provided all the necessary software and hardware support for the project at the high time.

I like to thank my colleagues in the software engineering group and the fellow graduate students for their spontaneous supports, comments and particularly their joyfulness that made my hard times smoother.

Finally, I make special mention of my family, specially my parents, who always motivated me for higher studies. They have supported and understood me every time and everywhere. Special thanks to them for everything they have done and given to me.

Declaration

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies make for study purposes, subject to normal conditions of acknowledgement

Abstract

With the growing popularity of web services the demand of highly reliable service based systems (SBS) is increasing. Formal verification and testing are performed to ensure the correctness of a system before it is deployed in a real environment. But the high complexity of complete fielded systems puts their effectiveness into questions. Runtime monitoring is the potential technique to cover the area not covered by formal verification and testing. This technique aims to assure the correctness of the current execution of a system. Substantial amount of research has been carried out in runtime monitoring to ensure the reliability of autonomous legacy software. However in service based system some significant complications arises as they focus on systems with no autonomous components, that make the approaches applied to monitor legacy software inadequate for service based system. In this thesis we present a framework for runtime monitoring of service based systems. We establish the necessity of introducing new types of inconsistencies beyond the classical inconsistencies that may occur during the execution of service based systems and develop reasoning mechanism to detect them at run time.

In the proposed framework, the properties to be monitored include: (i) *behavioural properties* of the co-ordination process of the service based system, (ii) *functional properties* that express functional requirements for the individual services of a service based system or groups of such services, (ii) *assumptions* regarding the behaviour of the service based system and its constituent services and their effects on the state of the system and (iii) *Quality-of-Service (QOS) properties* for the service based systems and its constituent services. All types of properties are expressed in a property specification language which is based on *event-calculus* [Sha99]. The *behavioural properties* to be monitored at run-time are extracted automatically from the specification of the co-ordination process of a service-based system in BPEL [Bpe03] while the other types of properties to be monitored must be specified by the providers of the system. These properties must be specified in terms of: (i) events that can be observed at run-time and correspond to either operation invocation and response messages or the assignment of values to global variables used by the co-ordination process of the system, and (ii) conditions over the state of the co-ordination process of the system and/or the individual services deployed by it. These restrictions ensure that property monitoring can be based solely on events which are generated by virtue of the normal opearation of the system without the need for instrumenting the individual services deployed by it. The property specification language that is used by this framework is a first-order logic language that incorporates special predicates to signify assertions about time and, to this end, it provides a very expressive framework for specifying properties of service based system, which may include temporal characteristics.

At run-time, the framework deploys an *event receiver* that catches events which are exchanged by the different services and the co-ordination process of the system and stores them in an event database. This database is accessed by a *monitor* that can detect different types of violations of properties. These types are: (i) violations of functional properties and quality-of-service properties by the recorded behaviour of the service based system, (ii) violations and potential violations of behavioural properties, functional properties and quality-of-service properties by the expected system behaviour, and (iii) unjustified and potentially unjustified actions which the system has taken by wrongly assuming that certain pre-conditions associated with the undertaken actions were satisfied at run-time. The detection of these types of violations is fully automatic and is based on an algorithm that has been developed as a variant of algorithms for integrity constraint checking in temporal deductive databases [Ple93, Cho95]. We have implemented a prototype of the proposed monitoring framework and showed the effectiveness of the monitoring prototype through several case studies.

Chapter One

Introduction

1.1 Overview

This thesis presents a framework that we have developed to support the runtime monitoring of service based software systems, i.e., systems which are implemented as compositions of web-services, against formally specified functional and quality of service requirements. Runtime monitoring provides a means of verifying dynamically the correctness of the actual behaviour of a system against a specification of how it should behave. The thesis provides a formal foundation of the monitoring framework that it proposes, describes the architecture and implementation of a prototype tool that we have developed to implement the framework, and investigates the effectiveness of the framework and its implementation by presenting the results of a set of experiments that we have conducted to evaluate them. The main contribution of this thesis is the provision of a novel framework for the runtime monitoring of service based software systems.

In the reminder of this introductory chapter, we give a comparative overview of different techniques that have been developed to assure the correctness of software systems, contrasts them with runtime monitoring and argues that runtime monitoring of service based systems is necessary in order to ascertain their correctness. The chapter concludes with a summary of the main contributions of the monitoring approach that is presented in this thesis.

1.2 Formal System Verification and Testing

Since the invention of microcomputers, the use of software has grown rapidly and now it has become a part of our everyday life. In fact now software is used to control much more safety critical equipment such as airline controllers, radiation controller or nuclear reactor, rather than simple audio or TV set. In most of these cases software failure can have serious consequences and even can devastate human lives and properties [Jac06] [Ari96] [Lev93]. Thus, there has been long standing and active research that is aimed at developing theories

and techniques that guarantee the correct execution of software systems including formal verification [Col98] [Tra99] and testing [IPL03] [Pan99]. Formal verification and testing can be applied to ensure the correctness of software systems during their development and before they are deployed in a real environment. This very fact, however, puts the effectiveness of such techniques into question as often it is impossible to anticipate all the possible conditions which may arise during the actual operation of a software system during its development and carry out verification and testing activities that can verify the behaviour of a system in all these conditions [Col98] [Whi00]. This particularly true for systems with decentralised architectures that may deploy dynamically evolving components such as the service based systems which constitute the focus of this thesis. Another factor that makes it unsuitable to verify a system prior to deployment is high complexity of complete fielded systems puts their effectiveness into questions.

1.2.1 Static Formal Verification

The objective of static formal verification is to prove formally that the behaviour of a software system is correct with respect to some *correctness criteria*, given a formal specification of the behaviour of this system. Static formal verification comes in two flavours: static verification by theorem proving [Fit96] and static verification by model checking [Cla94].

1.2.1.1 Model Checking

In model checking a software system is modelled as a finite state machine and the behavioural specification of the software is usually expressed in some temporal logic. These specifications are checked by a model checker. Typically, a model checker explores all possible states of the model and checks whether the properties in question are satisfied [Cla94] [Cla96]. The main model checkers that have been developed for static software system verification include: Spin [Hol97], SMV [Mcm92] and SLAM [Bal02].

The most significant advantage of model checking is that it is completely automatic and fast. On the other hand the main disadvantage of model checking is that it is not scalable to large system models and suffers from state space explosion problem [Eli06] [Cla96].

1.2.1.2 Theorem Proving

In theorem proving both the software model and the behaviour specification are expressed as formulas in some formal system, which defines a set of axioms and a set of inference rules. Theorem proving checks if each of the behavioural properties that have been specified for the system is a logical consequence of the formulas that represent the software model. To prove this logical consequence, theorem proving applies the axioms and the rules of inference of the formal system [Fit96] [Cla96].

The main advantage of theorem proving over model checking, is that it offers better scalability to deal large systems [Cla96]. The main disadvantage of theorem proving is that usually theorem prover requires human interaction. Also the computational complexity of theorem proving is high [Eli06] [Cla96].

1.2.1.3 Advantages and Disadvantages of Static Formal Verification

Static formal verification is used in the early software development process and helps developers gain a better understanding of a software system by revealing design flaws, inconsistencies, and ambiguities of a model. This practice can reduce the length of expensive and time consuming testing and debugging phases in software development. Despite these promises and the myth that static formal verification can guarantee that software is perfect, static formal verification still remains one of the most contentious areas in software engineering. This is because, in reality static formal verification can not guarantee that a software system is impeccable [Bow95] [Hal90]. Some of the most dominant factors that limit the applicability of static formal verification are:

- Static formal verification relies on the existence of a formal specification of the system. This specification has to be accurate and complete or otherwise static formal verification cannot be effective. Accuracy and completeness, however, are difficult to achieve in any form of modelling including formal specification due to the inherent complexity of the software systems which are being modelled and the environments which they operate in. In reality, formal specifications of software systems make implicit assumptions about a software system and its environment that limit significantly the completeness and accuracy of the results of static formal verification techniques [Hal90][Kim01a].

- Another inherent limitation of static formal verification arises due to the fact that even if a software specification model can be guaranteed to be accurate and complete there is no guarantee that the software system which will implement this model will be a correct and truthful implementation of the specification. Often this is not the case as implementation typically entails much more detail than a specification [Cop03] [Kim01a]. For example, in a model a data structure can be assumed to have infinite storage, but in implementation it should be restricted to some fixed size.

- The lack of scalability of static formal verification techniques is another point of concern. More specifically these techniques are known to have exponential time or space complexity and therefore while they can cope with relatively small system specification models they do not scale well with large models [Geo03][Cla96][Kim01a].

- Static formal verification may increase the cost of software development process and may delay the development process as its application often requires the use of large amount of computational resources and software developers with advanced skills and training in the field of formal methods [Geo03][Cla96][kim01a].

1.2.2 Testing

Testing is the process of analyzing software to identify and remove flaws thereby ensuring its correctness and quality [Har00]. This is achieved by executing several test cases to verify that a given program satisfies certain requirements. Test cases are constructed by experienced and highly qualified programmers (testers) to provide a good coverage of system operational scenarios and attempt to catch as many errors as possible. Software Testing requires the tester to have a close look at the source code and/or requirements specification and try to produce executions that will manifest the commonly recurring errors [Whi00].

1.2.2.1 Advantages and Disadvantages of Testing

Testing is performed to ensure the correctness of software, but it is less effective than specification verification, as it can not prove mathematical validity of the system [Ber03]. Moreover the errors detected by testing for a specific test case can not be generalised [Har00]. In addition to these limitations testing suffers from some other limitations including:

- Often testing is carried out in a well-known environment and the tester may be biased by the previous experience which may cause construction of test cases that may not cover all scenarios of a real environment, thus creating scope for bringing a system to operation without having tested in a significant spectrum of test cases. There have been long attempt to address this issue by automating software testing and test case generation. But the state of the art of automation of software testing still not able to handle this issue completely [Kan06] [Pra05].

- In some cases testing is completely unsuitable for checking some required properties [Hal90]. Consider, for example, *liveness* property, that is the property requiring that, "program execution eventually reaches some desirable state" [Owi82]. Such property in principle can never be verified by testing.

- Testing is less effective in verifying concurrent multi-threaded software systems [Kla04]. This is because, due to the non-deterministic nature of multi-threaded software, the occurrence of deadlocks and race conditions which may appear in this type of systems can not be predicted. Thus traditional testing techniques are inappropriate to capture errors related to these phenomena.

1.3 Runtime Monitoring

Runtime monitoring is the activity of determining at runtime whether a software system satisfies the formal requirements specification that is set for it. This process can be used to convince users that the run time behaviour of a system is compliant with its formal requirements in real conditions unlike testing and formal static verification.

Runtime monitoring takes a formal requirements specification for a software system as input and checks whether traces of events which are captured during the operation of this system at runtime is compliant with the specification [Del04]. This process primarily focuses on the evaluation of certain aspects of formal specification, for example the ordering of and other temporal relationships between events and can identify violations of the requirements or instances of unanticipated behaviour of a software system. Following the detection of such cases, alerts can be generated for fixing the behaviour of the system either manually or automatically (a system may have been designed with the capability to take recovery actions when unexpected behaviour occurs [Fea98]).

1.3.1 Advantages and Disadvantages of Runtime Monitoring

Runtime requirements monitoring has been proposed as a verification approach that can take into account the real conditions which arise during the operation of a system and test its behaviour against them [Del04]. The benefits of runtime monitoring over static formal verification and testing could be summarised as follows:

- Runtime monitoring checks the real implementation of a system as opposed to a specification that might not have been faithfully realised by an implementation.

- Runtime monitoring checks a system under real conditions that cannot be necessarily modelled in static models. For example the complete modelling of communication services (e.g. telecommunication service or Internet service) in static models is not practically possible [Die99].

- Static formal verification and testing attempt to ensure that all possible executions of the software satisfy desired properties. This leads to a state space explosion in the checking process that limits the scalability of these approaches. On the other hand, runtime monitoring focuses on a particular execution of the system, thus it may scale well for large systems.

- Unlike testing, which considers only a predefined set of inputs, runtime monitoring can potentially take into account all possible combinations of inputs. Thus it has the ability to reveal faults for rare and unexpected states which can not be addressed during testing.

- Runtime monitoring can be used to check certain type of properties that can not be checked in static verification. For example static verification is not suitable for checking properties related to real-time constraints, memory usage or concurrency [Cra93].

In spite of the above mentioned strengths of runtime monitoring over formal verification and testing, runtime monitoring has its own disadvantages:

- In some cases runtime monitoring may introduce undesirable side effects (e.g. lower performance) to the software being monitored [Urt02]. This may happen depending on the mechanism used to generate runtime events from the software and this is avoidable with careful design.

- Runtime monitoring can not guarantee the correctness of future executions of the software. More specifically, although runtime monitoring can detect faults in the current execution of the software, it can not ensure that the same fault would not occur in the next execution of the system. However it should be appreciated that the purpose of runtime monitoring is not to make a software faultless, but to detect runtime faults and help the designer/developer to make the software faultless.

1.4 Runtime Monitoring of Service Based Systems

As with classic software the objective of runtime monitoring of service based software systems (SBS systems) is to verify whether a web service based system is exhibiting the expected behaviour.

The key differentiating aspect of service based systems over traditional software is the highly distributed, decentralised and dynamically evolving nature of these systems which arises due to the deployment of web services by such systems. A web service is an autonomous, self-describing modular program which is, through standard XML messaging, accessible over a network, such as the Internet or an enterprise intranet, based on its standard transport protocols. The deployment of a web service is facilitated by a specification of the interface of a service, known as *service description that* specifies the different operations which are provided and can be invoked in the service. This description is expressed in XML (typically using the WSDL standard [Wsd04]) and provides all the necessary details, such as transport protocols, location of the service and message format, to interact with the service. Since a web service interface hides the implementation details of the service in a service based system it can be used independently of the hardware or software platform on which it is implemented and also independently of the programming language in which it is written. This aspect of web services enables the web service based systems to be loosely coupled.

Monitoring the behaviour of service based systems at runtime presents some special challenges in comparison with the runtime monitoring of classical software systems. This is because

- The most powerful feature of service based systems is that they allow the composition of heterogeneous web services offered by different vendors to provide

value-added services to their customers. In traditional software systems or distributed systems software components are fixed and hard-wired together or tightly coupled. This means that a monitor can be developed by considering a specific configuration of software components that provide a specific service for the system. Service based systems however may interact with other computational entities (web services) which may have been developed by different vendors, written in different programming languages and running on heterogeneous computing platforms. This difference increases the complexity of the runtime monitoring of service based systems over their traditional counterparts. For example, in service-based systems, the failure of specific services to function as expected may lead other system components to make incorrect assumptions about the state of the system (e.g. the absence of a message confirming the update of some data in one of the system's services does not necessarily mean that these data have not been updated. See Section 2.4 in Chapter 2 for a specific example). Consequently, components may take actions, which may be compliant with the requirements but would not have been taken if the correct state of the system was known to them.

- Service based systems often have the ability to evolve dynamically and may change its behaviour at runtime. This may happen at least in two ways. Firstly, the composition process of a service based system accesses a web service through its interface and is unaware of the implementation details of the service. Therefore any change in the implementation of any web services in the composition, such as update of performance measures or addition of new functionality, is beyond anticipation at design time and deployment time. Secondly, in a service based system a malfunctioning web service can be replaced by a new web service at runtime. This possibility of reconfiguration of service based system and evolving nature of web services must have significant impact on runtime monitoring (e.g. a monitoring property may prove invalid with respect to the new configuration of the service based system).

- In traditional software systems (even distributed ones) which are based on a fixed set of software components, a monitor may exercise a great deal of control over the system within the infrastructure on which it is deployed. Service-based systems, however, receive key services from outside providers (web services) and they are designed not to be dependent on any collective computing entity. So web services must be treated as strictly autonomous units and the provider of the service based system can not be assumed to have ownership of the code of the individual services

that it deploys. This impacts upon the ability to generate internal events from the individual web services which are deployed by service based systems.

- Monitoring of non-functional requirements for service based systems should be of great importance. For example response time, one can't expect a customer to wait indefinitely long time to receive the service. Because of the loosely coupled distributed nature of web services monitoring of non-functional requirements could be critical. In service-based systems, the specification and checking of non functional requirements must take into account the time required for the communication between the interacting services. This time, however, is not negligible as it is typically assumed in requirement specifications of service based systems, and may vary depending on the physical distribution of the services on different processors and/or network communication delays.

1.5 The Monitoring Approach of this Thesis

This thesis presents a framework that supports the monitoring of requirements for service-based systems, which are implemented in BPEL [Bpe03], that is an executable language for specifying service composition workflows, and provides the foundation for addressing the critical issues in runtime monitoring of service based systems identified in Section 1.4. In this framework, requirements are specified in terms of behavioural properties of a service-based system, which are automatically extracted from the specification of the composition and the interactions of the individual services in it. The developed framework also supports system providers to specify (i) functional properties that express functional requirements for the individual services of a service based system or groups of such services, (ii) assumptions about the environment of the system, the actions of the agents in it, or the individual services of the system and (iii) quality of service (QoS) properties, which express the quality requirements of individual services or group of services. These functional properties, assumptions and QoS properties are specified in terms of event occurrences and/or conditions about the values of state variables that have been identified from the compositional specification of the system during the extraction of the behavioural properties. At run-time, the proposed framework obtains information about the state of these variables and event occurrences by catching events which are exchanged between the individual services and the co-ordinating component of the system without requiring the modification of the code that implements these services. This framework aims to monitor both functional and non-functional requirements that should be satisfied by service-based software systems and

incorporates hybrid reasoning mechanisms, namely deductive and abductive reasoning for identifying violations of these requirements.

1.6 Contributions

The main contributions of the approach that is presented in this thesis and the framework that implements it are:

- **Implementation of a non intrusive monitoring framework**

In this research work we develop a monitoring framework that is suitable for runtime monitoring of service based system. Unlike the other approaches that we are aware of, our framework applies non intrusive approach to monitoring. The term "non intrusive monitoring" here signifies a form of monitoring that is carried out by a computational entity that is external to the system that is being monitored, is carried out in parallel with the operation of this system and does not intervene with this operation in any form. The framework that we present in this thesis is non intrusive as it is based on events which are captured during the operation of a service based system without the need to instrument its composition process or the code of the services that it deploys and is performed by a reasoning engine that is separate from the system that is being monitored and operates in parallel with it. This approach is motivated from the fact that in case of service based system the service provider may not have ownership of the code of the individual services that it deploys, therefore monitor would not have any control over the individual services (see Section 1.4).

- **The detection of types of property deviations which are not detected by other monitoring approaches and tools**

It is discussed in Section 1.4 that because of loose coupling and distributed nature, in service based system the failure of specific services to function as expected may lead other system components to make incorrect assumptions about the state of the system. For example the absence of a message confirming the update of some data in one of the system's services does not necessarily mean that these data have not been updated, and the evidence of the absent message can be inferred by logical reasoning. To cover these possibilities our framework makes a broad distinction to the type of the events which are used in order to detect property deviations. These events may be of two types: (1) recorded events which have been captured during the operation of the system at runtime or (2) derived events which are generated from

recorded events by logical reasoning (e.g. deduction/abduction). The use of events of these two types also affects the characterisation of property deviations. More specifically, if monitoring is based only on recorded events, it can detect only deviations which are evidenced by violations of specific properties by these recorded events. If, on the other hand, monitoring is based on both recorded and derived events, then the framework can also detect (a) inconsistencies which arise from the expected system behaviour, (b) cases of unjustified system behaviour, (c) possible inconsistencies evidenced from the expected system behaviour, and (d) possible cases of unjustified system behaviour. We also devise the appropriate reasoning mechanisms to detect the occurrence of these new types of property deviations during the operation of service based systems. To the best of our knowledge while other approaches support the detection of property deviations based on recorded events only, they do not support the detection of deviations based on derived events.

- **Property specification language**

We define a language to specify requirements of service based systems. This language has its formal foundation on Event Calculus [Sha99] and it enables the specification of monitorable properties using full first-order logic formalism as well as conditions about time. However our language allows the use of internal and external operations in formulas that can perform complex computations. This makes our language expressive enough for specifying a wide spectrum of monitorable properties including behavioural properties, functional properties and QoS properties. Also, our language defines special types of events that can be encountered in service based systems (e.g. callbacks). These types of events are defined as part of standard event calculus. Furthermore, we define a schema to express a formula specified in our language in XML that makes our language applicable to other standards as a means of specifying requirements.

- **Implementation of a prototype supporting the monitoring of service based systems implemented in BPEL**

We have developed a prototype of the monitoring framework. This prototype provides supports for automatic monitoring of service based systems implemented in BPEL, i.e. (i) it automatically extracts the behavioural properties to be monitored from the service composition specification (ii) it allows users to define additional functional properties, QoS properties and assumptions about the service based system and (iii) it monitors the target service based system automatically following the behavioural properties, functional properties, QoS properties and assumptions. The prototype can be used as a stand alone

monitoring tool as well as it can be deployed as a web service. In either case the tool is easy to set up and provides maximum user flexibility.

1.7 Outline of the Thesis

This thesis is organised into seven chapters including this chapter.

Chapter 2 provides a literature review. In this chapter, we present the main technological platforms and standards which are used by service based systems. We also investigate previous work in the area of runtime requirements monitoring and identify open research issues/limitations of the existing approaches when it comes to the monitoring of service based systems. These limitations are identified as they constitute the factors that have motivated the development of the approach that is presented in this thesis and the framework that we have developed to implement it.

Chapter 3 introduces and explains the policy specification and property specification language used in the monitoring framework. It also presents the motivation behind the use of event calculus to represent properties in our framework.

Chapter 4 defines the types of property deviations that may occur during the execution of service based systems and can be detected by our framework. Subsequently it specifies the monitoring scheme that is used to detect these types of property deviations. The description of the monitoring scheme includes the algorithms used for the detection of property deviations, and a formal analysis of the complexity, soundness and completeness of these algorithms.

Chapter 5 discusses the implementation of a prototype for the proposed monitoring framework. It also demonstrates the use of the prototype using monitoring scenarios.

Chapter 6 presents the result of an experimental evaluation of the monitoring framework which has been based on two separate case studies. The first of these case studies is based on monitoring a simulated BPEL process. The second case study is based on monitoring a real BPEL process that deploys external web services provided by different providers over the Internet.

Finally, Chapter 7 concludes with the summary of the monitoring framework that is described in this thesis, the contributions of the research underpinning it to the state of the art in this area, and recommendations for further work.

Chapter Two

Runtime Verification of Service Based Software Systems – State of the Art

2.1 Overview

The purpose of this chapter is to give the readers necessary technical background on web services and to help understand the current trend of research in monitoring requirements of legacy software and service based systems. Section 2.2 covers the technical background on web services. First we succinctly describe the web service architecture, and then we briefly present the standards that enable the web service architecture. We also present comparative discussion of the standards for composing web service based systems and the standards for specifying service level agreements of web service based systems.

Section 2.3 presents critical analysis of current research in the requirements monitoring of legacy software and service based systems. For the ease of discussion we introduce a general requirements monitoring framework that has been used to compare different approaches found in the literature. Section 2.4 identifies open research issues/limitations of the existing approaches to handle the monitoring of service based systems that motivates this research. In Section 2.5 we introduce our approach to address the issues raised in Section 2.4.

2.2 Web Service Technologies and Standards

In this section we focus on the Web Service architecture and briefly present the Web-Service enabling technologies and standards including Simple Object Access Protocol (SOAP) [Soa03], Web Service Description Language (WSDL) [Wsd04] and Universal Description Discovery and Integration (UDDI) [Udd03]. We also present a comparative summary of Web service composition languages.

2.2.1 Basic Concepts and Definitions

In this section we present the definitions of the main concepts that are used to describe service centric system through out this thesis. These concepts are taken from W3C Web Services Glossary [Wsg04].

Service: A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of service provider and service requester. To be used, a service must be realized by a concrete provider agent.

Web service: A web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards.

Service Provider: The person or organization that is providing a web service.

Service Requestor: The person or organization that wishes to use a service provider's web service.

Provider Agent: A software agent that is capable of and empowered to perform the actions associated with a service on behalf of its owner i.e. service provider.

Requester Agent: A software agent that wishes to interact with a provider agent in order to request that a task be performed on behalf of its owner i.e. service requestor.

Discovery Agency: Discovery agency is a set of service descriptions where service providers publish their service descriptions. Discovery agency enables service requestors (or service providers) to search the set of service description to find a specific service.

Choreography: Web Services Choreography concerns the interactions of services with their users. Any user of a Web service, automated or otherwise, is a client of that service. These users may, in turn, be other Web Services, applications or human beings. Transactions among Web Services and their clients must clearly be well defined at the time of their execution, and may consist of multiple separate interactions whose composition constitutes a complete

transaction. This composition, its message protocols, interfaces, sequencing, and associated logic, is considered to be a choreography.

Orchestration: An orchestration defines the sequence and conditions in which one Web service invokes other Web services in order to realize some useful function. i.e., an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal.

2.2.2 Web-Service Architecture

The W3C consortium defines the basic web service architecture as a "stack" of relationships among various technologies and components [Wsa02, Wsa04]. The web service architecture defines an interaction between software agents capable of performing three roles: (a) the service provider, (b) the service requestor, and (c) service discovery agency. The interactions involve service publishing, finding and binding operations. Figure 2.1 illustrates the roles in the web service architectures [Wsa02]

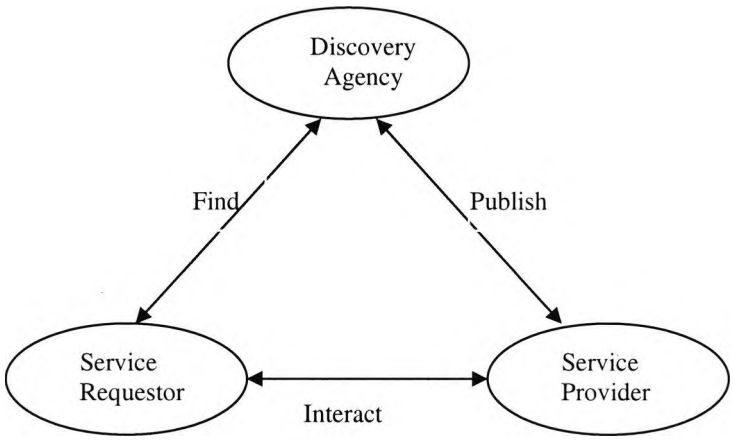


Figure 2.1: Basic web-service architecture

In a typical scenario (Figure 2.1) a service provider hosts a network accessible software module, defines a service description for the software module and publishes it to a requestor or a service discovery agency. The service requestor uses a 'find' operation to retrieve the service description locally or from the discovery agency and uses it to bind with the service provider and interact with the web-service implementation. In a web-service architecture, a software agent can perform one or more of the above roles, such as requestor or provider only, both requestor and provider, or as requestor, provider and discovery agency.

The W3C consortium has also defined an extended web service architecture [Wsa02, Wsa04]. This architecture extends the technologies and components defined in the basic web-service architecture by incorporating some additional features and functionality. Figure 2.2 shows a simplified view of the stack of layers contained in the extended web-service architecture described in [Wsa02, Ws04]. In this simplified view we present only the layers pertinent to this research. Figure 2.3 shows a mapping between the basic web service architecture and the extended web service architecture. The layers of the extended web service architecture shown in Figure 2.2 are described in the following.

Discovery Agency	Discovery
	Publication
Description	Service Level Agreements
	Composition
	Presentation
	Policy
	Implementation Description
	Interface Description
Wire	Extensions
	Packaging
	Transport

Figure 2.2: Extended web-service architecture (Simplified view)

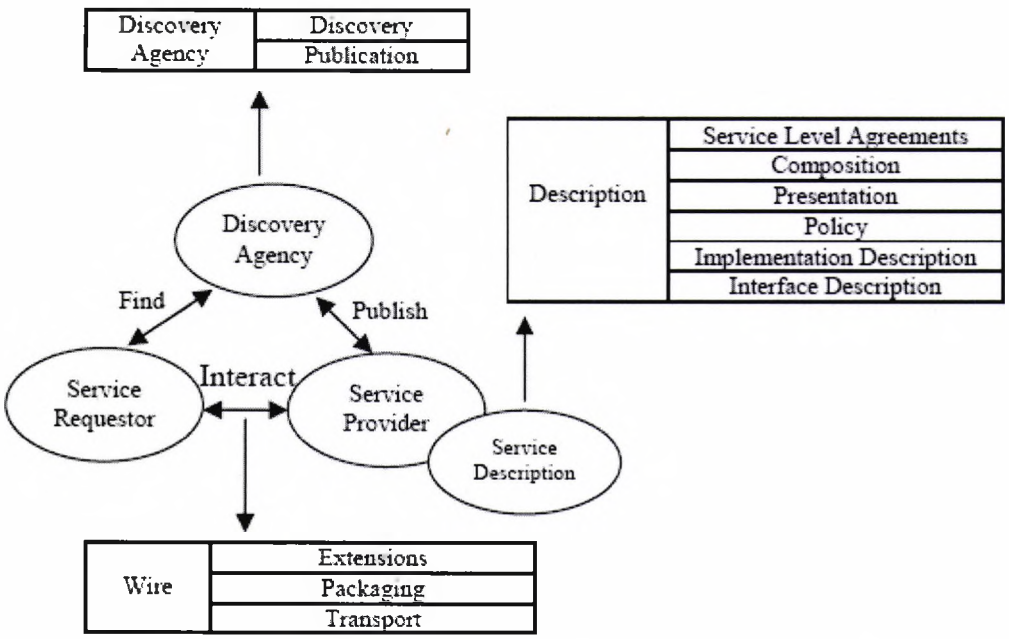


Figure 2.3: Mapping between basic web service architecture and extended web service architecture

2.2.2.1 Wire Layer

This is the physical layer of the architecture and deals with the actual physical exchange of information between any of the roles in the basic architecture. This layer can be divided into three sub-layers, namely the *transport*, *packaging* and *extension* layers.

The transport layer provides the protocol that will be used for the network communication. The web-services that are publicly available on the Internet use commonly deployed network protocols such as HTTP [Htt99]. HTTP is a set of rules that define how messages (e.g. text, images, sound) are formatted and transmitted over the Internet. Other Internet protocols may be supported including SMTP [Smt82] and FTP [Ftp80]. SMTP is a set of rules that is used in sending e-mails between mail servers over the Internet and FTP is a set of rules for exchanging files over the Internet. Intranet domains may use platform or vendor specific protocols such as CORBA [Cor04].

The packaging layer represents the technologies that may be used to package the information being exchanged between services. SOAP is a simple and lightweight XML-based protocol for creating structured data packages that can be exchanged between network applications and it has been widely adopted as the basis for a web-service message packaging protocol. A brief introduction of SOAP is presented in Section 2.2.3.1.

The extensions layer offers a way to attach additional information to web-service messages such as message routing information, message transaction context.

2.2.2.2 Description Layer

The description layer includes a collection of description documents, which are typically expressed in XML. The service provider provides all the specifications for invoking the web-service through the service description document. In the simplified web-service architecture shown in Figure 2.2, the description layer incorporates only six sub-layers of different types of web-service description documents, namely: (a) interface descriptions, (b) implementation descriptions, (c) policy descriptions, (d) presentation descriptions, (e) composition description and (f) service level agreements description. The interface and implementation descriptions are the minimum parts of a web-service description which are necessary to enable remote invocations of a web-service. The service interface description is similar to the definition of an abstract interface in programming languages e.g. Java [Jav94] and may have multiple

concrete implementations. The service implementation description contains information about the concrete implementation of a particular service interface, including the location of the implementation. The Web Service Description Language (WSDL) is used for the construction of a base level description of a web-service interface. WSDL is an XML schema for specifying such descriptions which is further discussed in Section 2.2.3.2. A policy description contains a set of assertions or rules that applies constraints on the behaviour of a service. For example policy description can be used to define security policies, quality of service attributes and management requirements. A presentation description defines different ways of rendering a web-service on a variety of computational devices (e.g., desktops, phones, PDAs). The composition description defines the programmatic relationships between web services where more than one web services are involved to complete a multi-step business interaction. A service level agreement description contains the contractual agreements regarding the guarantees of a web service between service provider and service consumer.

2.2.2.3 Discovery Agency

The discovery agency layer in Figure 2.2 denotes a searchable repository of service descriptions. This repository enables service providers to publish their service descriptions and service requestors to find services and obtain binding information (i.e., information for calling the service). The simplest form of discovery is static discovery where the requestor caches the service description at design time by retrieving the description from a local file or local service description repository and the cached description is used at runtime. On the other hand, in dynamic discovery, the requestor discovers the service at design time or runtime accessing a local WSDL registry or a public/private WSDL registry such as Universal Description Discovery and Integration (UDDI). In case of dynamic discovery the service registry should support a query mechanism that enables the requester to find a service by different parameters including type of interface (based on a WSDL template), properties (such as QoS parameters), and the taxonomy of the service. UDDI is a standard application programming interface that is used to provide access to service registries (see Section 2.2.3.3 for more details).

2.2.3 Basic Web-service Standards

2.2.3.1 Simple Object Access Protocol (SOAP)

SOAP is an XML-based protocol that governs the encoding, exchanging and processing of messages in inter application communication. SOAP is intended for exchanging structured data between network applications in a decentralised, distributed environment, independently of the underlying platform. SOAP can be used in combination with a variety of other network protocols such as HTTP, SMTP or FTP.

2.2.3.1.1 SOAP Message Construct

A SOAP message is an XML document that consists of three basic parts: an *envelope*, a *header* and a *body*, as shown in Figure 2.4. These basic building blocks of a SOAP message serve the following purposes:

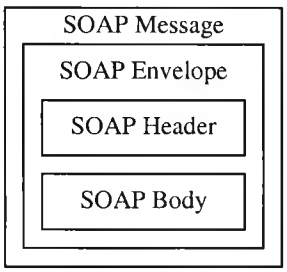


Figure 2.4: SOAP message structure

- The *Envelope* element identifies the XML document as a SOAP message. It primarily defines the namespaces which are used to define the elements and attributes in the rest of the message. This element is necessary in a SOAP message.
- The *Header* element contains auxiliary application specific information, (e.g. user authentication information), that needs to be exchanged between the participating applications. A SOAP message may contain zero or one headers. If present, the header should be the first element inside the Envelope.
- The *Body* element contains the XML information that is exchanged by the message. This part of a SOAP message may have an additional sub element, called *Fault*, providing information about errors that may occur while processing the message.

A SOAP message skeleton is shown in Figure 2.5.

```
<?xml version="1.0"?>
<Env:Envelope
xmlns:Env="http://www.w3.org/2002/12/soap-envelope"
Env:encodingStyle="http://www.w3.org/2002/12/soap-encoding">
        <soap:Header>
          ...
        </soap:Header>
        <soap:Body>
          ...
                <soap:Fault>
                  ...
                </soap:Fault>
        </soap:Body>
</Env:Envelope>
```

Figure 2.5: A SOAP message skeleton

2.2.3.2 Web Service Description Language (WSDL)

The Web Services Description Language (WSDL) is an XML based language for describing web services. WSDL specifies the location of the service and the operations (methods) that it exposes. The operations and messages are described abstractly and then bound to a concrete network protocol and message format to define a location.

WSDL describes web services starting with the messages that can be exchanged between the service provider and the requestor of the service. A *message* consists of a collection of typed data items. An operation is a combination of *messages* to define the message exchange pattern supported by the web service. A collection of operations is known as *PortType*. A *service* is defined as a collection of *Ports*, each of which is an implementation of a *PortType*. A *Port* includes all the concrete details needed to interact with the *service*, i.e. binding and location.

<table>
<tr><td rowspan="2">Service
Implementation
Definition</td><td>Service</td></tr>
<tr><td>Port</td></tr>
<tr><td rowspan="4">Service
Interface
Definition</td><td>Binding</td></tr>
<tr><td>PortType</td></tr>
<tr><td>Message</td></tr>
<tr><td>Type</td></tr>
</table>

Figure 2.6: WSDL document elements [Wsa02]

In the extended web service architecture shown in Figure 2.2, the service implementation definition together with service interface definition forms a complete service definition. Figure 2.6 shows the WSDL elements that become part of the service interface and service implementation description documents.

WSDL definitions are represented in XML by one or more WSDL information sets. An information set is a *definition* element. A WSDL information set contains representations for a collection of WSDL components.

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri">
    <import namespace="uri" location="uri"/> *
    <wsdl:documentation .... /> ?
    <wsdl:types> ?
        <wsdl:documentation .... /> ?
        <xsd:schema .... /> *
    </wsdl:types>
    <wsdl:message name="ncname"> *
        <wsdl:documentation .... /> ?
        <part name="ncname" element="qname"? type="qname"?/> *
    </wsdl:message>
    <wsdl:portType name="ncname"> *
        <wsdl:documentation .... /> ?
        <wsdl:operation name="ncname"> *
            <wsdl:documentation .... /> ?
            <wsdl:input message="qname"> ?
                <wsdl:documentation .... /> ?
            </wsdl:input>
            <wsdl:output message="qname"> ?
                <wsdl:documentation .... /> ?
            </wsdl:output>
            <wsdl:fault name="ncname" message="qname"> *
                <wsdl:documentation .... /> ?
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:serviceType name="ncname"> *
        <wsdl:portType name="qname"/> +
    </wsdl:serviceType>
    <wsdl:binding name="ncname" type="qname"> *
        <wsdl:documentation .... /> ?
        <-- binding details --> *
        <wsdl:operation name="ncname"> *
            <wsdl:documentation .... /> ?
            <-- binding details --> *
            <wsdl:input> ?
                <wsdl:documentation .... /> ?
                <-- binding details -->
            </wsdl:input>
            <wsdl:output> ?
                <wsdl:documentation .... /> ?
                <-- binding details --> *
            </wsdl:output>
            <wsdl:fault name="ncname"> *
                <wsdl:documentation .... /> ?
                <-- binding details --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="ncname" serviceType="qname"> *
        <wsdl:documentation .... /> ?
        <wsdl:port name="ncname" binding="qname"> *
            <wsdl:documentation .... /> ?
            <-- address details -->
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figure 2.7: Overview of WSDL schema

Figure 2.7 shows an overview of the WSDL schema defined by W3C consortium. Here we present only the part of WSDL schema that is pertinent to this research and we explain the selected elements of the WSDL schema with a simple example. In this figure

- *?* identifies that the element or attribute appears 0 or 1 time at this position.
- * identifies that the element or attribute appears 0 or more times at this position.
- *nmtoken* is a placeholder for any name value corresponding to the NMTOKEN data type from XML schema specification defined by W3C. NMTOKEN stands for *name token*, which is any string of text made of legal XML name characters where legal XML name characters include letters, digits, ideographs and the underscore, hyphen, colon, and period.
- *qname* is a placeholder for any name value corresponding to the QName data type from XML schema specification defined by W3C.

Figure 2.8 shows, a sample web service implemented in Java. This service has only one method, called *add* that takes two *doubles* as input and returns their sum as output.

```
public class AddTest{
    public double add(double op1, double op2){
       return op1 + op2;
    }
}
```

Figure 2.8: Example of a web service written in Java

Figure 2.9 shows the interface and implementation description for the service presented in Figure 2.8 written in WSDL.

The document starts with XML header, followed by a `<wsdl:definitions>` element that contains common namespace declarations and encloses the remaining WSDL document elements. The `<wsdl:message>` elements define the data exchanged during communication with the service. The message *addRequest* represents an incoming request and consists of two parts of type `xsd:double` defined by the `<wsdl:part>` element of the message. These two parts correspond to the input parameters of the addition request (i.e., op1 and op2). The message *addResponse* represents the outgoing response and has one part of type `xsd:double` (i.e., addReturn). The `<wsdl:portType>` element contains all operations provided by the service *AddTest*. These operations are defined by the sub-elements `<wsdl:operation>` of the `<wsdl:portType>` element. As shown in Figure 2.8, *AddTest* has only one operation–the operation *add*. Each `<wsdl:operation>` element is

defined using the incoming and outgoing messages defined in the WSDL definition. The operation *add*, for example, is defined using the messages *addRequest* and *addResponse*.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://tempuri.org/services/calc"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:impl="http://tempuri.org/services/calc"
                  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:message name="addRequest">
      <wsdl:part name="op1" type="xsd:double"/>
      <wsdl:part name="op2" type="xsd:double"/>
   </wsdl:message>

   <wsdl:message name="addResponse">
      <wsdl:part name="addReturn" type="xsd:double"/>
   </wsdl:message>

   <wsdl:portType name="AddTest">
      <wsdl:operation name="add" parameterOrder="op1 op2">
         <wsdl:input message="impl:addRequest" name="addRequest"/>
         <wsdl:output message="impl:addResponse" name="addResponse"/>
      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="AddTestServiceSoapBinding" type="impl:AddTest">
      <wsdlsoap:binding style="rpc"
            transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="add">
         <wsdlsoap:operation soapAction="http://tempuri.org/add"/>
         <wsdl:input name="addRequest">
            <wsdlsoap:body
                encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://tempuri.org/services/calc"
                use="encoded"/>
         </wsdl:input>
         <wsdl:output name="addResponse">
            <wsdlsoap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://tempuri.org/services/calc"
                use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="AddTestService">
      <wsdl:port binding="impl:AddTestServiceSoapBinding"
              name="AddTestService">
         <wsdlsoap:address location=
              "http://138.40.91.72:8080/wstk/services/AddTestService"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

Figure 2.9: Sample web service description in WSDL

The attribute *type* of the element `<wsdl:binding>` points to the port of the binding, in this case *AddTest*. The attribute *name* of the element `<wsdl:binding>` indicates the name

of the binding. The element <wsdlsoap:binding> is used to define the style and transport used. The *style* attribute specifies the message format to be used, which could be *rpc* or *document*. And the *transport* attribute specifies the network transport protocol to be used. In the above example the values of *style* and *transport* attributes are *rpc* and *SOAP* over *HTTP* respectively. For each operation (only one in this case) in the port there is a <wsdlsoap:operation> element.

The attribute *soapAction* of a <wsdlsoap:operation> element holds an URI which uniquely identifies the service among the deployed services on a particular SOAP server. The element <wsdlsoap:operation> contains the elements <wsdl:input> and <wsdl:output>, which correspond to the same elements of the *PortType* element. These elements specify how the input and output are encoded respectively (a SOAP encoding is used in this case). The element <wsdl:service> represents the web service as collection of port elements (only one in this case). Each <port> element associates a <binding> to a location i.e. the address information (a URI) to locate this service. This is a one-to-one association. In this case, the binding *AddTestServiceSoapBinding* has been associated to the location http://138.40.91.72:8080/wstk/services/AddTestService

2.2.3.3 Universal Description, Discovery and Integration (UDDI)

Universal Description, Discovery and Integration (UDDI) has been defined [Udd00] as a platform independent framework that enables organisations to describe their business and web services and discover services from other organisations. The UDDI framework uses two major information structures:

I. A set of specifications for distributed web-based information registries of web services. UDDI provides a data structure that can be used to specify a business and the service provided by it and an API specification to access the stored information.

II. A set of implementations of specifications (known as UDDI registry), accessible through network, that allows service providers to register information about the Web Services they offer so that other service providers or service requesters can find them. UDDI registries can be [Udd02]:

- **Public:** Other than the administrative functions, access to the registry data is open and public. Data may be shared with or transferred to other registries. Currently, there are two UDDI registries which have been provided for public access; one hosted by

IBM [Ud_IBM] and another hosted by Microsoft [Ud_MIC]. These repositories synchronise their contents regularly so that information entered into one registry is replicated to the other one so as to be available from it.

- **Private:** These are internal registries, set up behind firewalls that are isolated from the public network. Access to both the administrative features and registry data is secured and data can not be shared with other registries [Udd02].

- **Shared/Semi-Private:** These are registries deployed within a controlled environment. This environment provides controlled access to the outside world and allows sharing information with trusted outside partners only.

2.2.4 Orchestration and Choreography of Web Services

While in some cases web services may be deployed in an isolated form, in other it is difficult to find a specific single service that can fulfil a complex business requirement. In the latter cases, it may still, however, be possible to find combinations of isolated web services that can satisfy the requirement. In such cases, web service architectures provide a way to fulfil a complex business requirement by combining existing web services. Recent research in combining isolated web services has flourished in two closely related paradigms namely, *web service orchestration* and *web service choreography*.

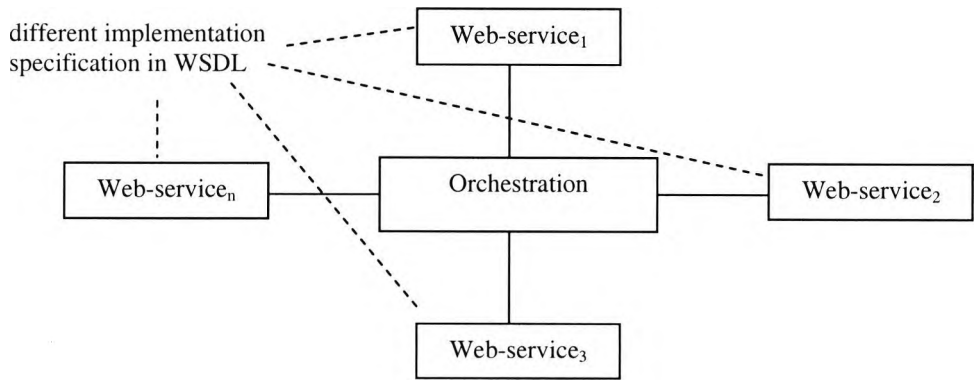


Figure 2.10: Web service orchestration

Orchestration refers to the definition of new service by combining existing web services. In other word orchestration defines an executable process that may interact with external web services at the message level including the logic and execution order of the interactions. Figure 2.10 shows the structure of web service orchestration.

Choreography is a description of the sequence of messages that is exchanged between multiple web services, i.e. it describes the logic under which a particular web service operation can be invoked where multiple web services are involved. Figure 2.11 shows the structure of web service choreography.

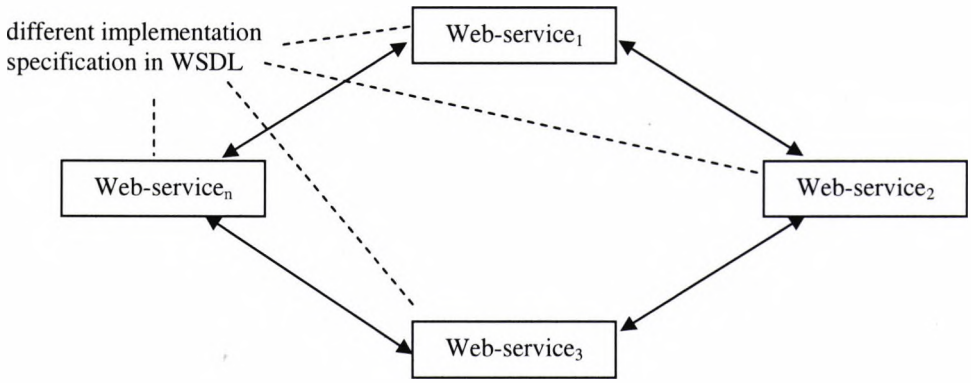


Figure 2.11: Web service choreography

Choreography differs from orchestration in that orchestration specifies interactions between services controlled by a single service, whereas choreography describes the observable interactions between services and their users not truly controlled by any single service.

Recently, several XML-based languages have emerged to express the logic of web service orchestration, including BPEL4WS [Bpe03], OWL-S [Owl04a] and web service choreography including WSCI [Wsc02], WS-CDL [Wsc05]. In the rest of this section we present a comparative summary of these web service orchestration and choreography languages.

2.2.4.1 Web Service Orchestration Languages

2.2.4.1.1 Business Process Execution Language for Web Services (BPEL4WS)

Two of the earliest languages developed to provide standards for web service composition were the Web Service Flow Language (WSFL) [Wsf01] that was developed by IBM and XLANG [Xla01] that was developed by Microsoft. In 2002, WSFL and XLANG were merged into a common language, called *Business Process Execution Language for Web Services* (BPEL4WS) [Bpe03]. BPEL (in the rest of this thesis BPEL4WS is referred to as

BPEL) integrated the concepts of WSFL and XLANG into a unified standard to support the modelling of executable and abstract processes that compose web services.

An abstract process in BPEL specifies the public message exchanges between web services but does not convey the internal details of the process flow. An abstract process is not intended to be executed. An executable process models the behaviour of the participant web services in a specific business interaction. In a typical scenario a BPEL executable process receives messages with processing requests from its participant services, invokes operations to perform specific computations in the same or other services and responds (with messages) back to the requestors.

BPEL provides a set of activities to model business processes. These activities are divided into two types, namely *basic activities* and *structured activities.*

Table 2.1: Summary of BPEL activities

Basic activity	**Description**
receive	This activity enables BPEL process to receive messages from its partner services.
invoke	This activity enables BPEL process to invoke an operation on a partner.
reply	This activity enables BPEL process to send a message to its partner.
assign	This activity is used to copy data from one variable to another variable in a BPEL4WS process
throw	This activity enables a BPEL process to signal internal faults.
wait	This activity allows a BPEL process to wait for a certain period of time
empty	This activity is used for no operation purpose in a process, for example to catch and suppress an internal fault.
Structured activity	**Description**
sequence	This activity specifies an ordered list of other activities that must be performed sequentially in the exact order of their listing
flow	This activity specifies two or more groups of other activities that should be executed concurrently. A flow activity completes when all of the groups of activities in it have completed. In a concurrent execution, it might be necessary to specify synchronisation dependencies between activities. Inside a flow, such dependencies are specified using *links*. Each link defines a *target* activity which cannot start before the completion of a *source* activity which is also defined by the link
switch	This activity specifies an ordered list of one or more conditional branches of groups of other activities which may be executed subject to the satisfiability of conditions associated with each branch
pick	This activity makes a composition process to wait for different events and then perform different activities associated with each of these events as soon as it occurs. A *pick* may also have a timer that can make it execute a different activity if none of the expected events happens within the time period specified by the timer
while	This activity is used to specify iterative occurrence of one or more activities as long some condition holds true

Basic activities support primitive functions such as communicating with web services (e.g., activities for invoking an operation in a web service or replying to a web service) and managing data (e.g. assignment of variable values). Structured activities provide control and data flow structures that enable the composition of basic activities into a business process. The latter activities support the specification of sequential and parallel execution of services, iterative execution of services, and dynamic branching. Table 2.1 presents a summary of the basic and structured activities offered by BPEL.

2.2.4.1.2 OWL-S

OWL-S [Owl04a] is an ontology for semantic markup of web services. The aim of the semantic web [Ber01] is to bring structure to the meaningful content of web resources which is easily processable by computers and this will make the web resources accessible by content rather than just by keywords. OWL-S is OWL (Ontology Web Language) [Owl04b] based Web Service ontology, which provides a core set of markup language constructs for describing the properties and capabilities of web services in an unambiguous computer interpretable form.

OWL-S intends to facilitate the automation of web service tasks including automated discovery, execution, interoperation, composition and execution monitoring. OWL-S models services using the ontology consisting of three parts:

(i) A *service profile* that describes the properties of a service, which are necessary for automatic discovery. These properties include the name, provider and a functional description of the service. The latter description specifies the input, output, preconditions and effects of the service. For example, in case of a ticket booking service input is the travelling destination, date and credit card number, output is the description of the booked ticket, precondition is 'credit card is valid' and the effect of the execution of the service is 'credit card has been charged'.

(ii) A *service model* that describes how to interact with a service considering it as a process. In OWL-S, a process can be an atomic process or a composite process. An atomic process is one that is directly invocable with appropriate messages and executed in a single step and returns a response. An atomic process is expressed by describing its inputs, outputs, preconditions and effects. Composite processes are composed of other composite or atomic processes. OWL-S provides a variety of flow control constructs, including sequential and parallel processing and conditional

iterations, to define composite processes. Table 2.2 presents a summary of OWL-S control constructs.

Table 2.2: Summary of OWL-S control constructs

Control constructs	Description
Sequence	*Sequence* specifies a list of processes to be executed in the given order.
Split	*Split* specifies a group of processes to be executed concurrently.
Unordered	Unordered defines a group of processes that must be executed, but not in any specific order
Choice	*Choice* allows a process to be executed from a group of processes.
If-then-else	*if-then-else* allows one of two processes to be executed based on the truth value of some condition.
Iterate	*iterate* allows a group of processes to be executed repeatedly.
Repeat-while/ Repeat-until	Both of these allow a group of processes to be executed repeatedly based on the truth value of some condition.

(iii) A *service grounding* that gives information on how to access a service. OWL-S extends WSDL to connect the abstract representations of *service profile* and *service model* to the concrete level of specification. For example a OWL-S atomic process corresponds to a WSDL *operation*, inputs/outputs of a OWL-S atomic process correspond to WSDL *message* and OWL-S relies on WSDL binding constructs to specify these.

2.2.4.2 Web Service Choreography Languages

2.2.4.2.1 Web Service Choreography Interface (WSCI)

Web Service Choreography Interface (WSCI) [Wsc02] is an interface definition language for describing the overall collaboration between web services by specifying the flow of messages exchanged among the involved web services. WSCI specifies the external observable behaviour between web services within the context of the specific interactions rather than the internal definition of service behaviour. Thus, WSCI does not support the definition of an executable service coordination process as BPEL.

WSCI provides a rich collection of activities to model choreography of message exchanges between web services. Activities can be *atomic* and *complex*. Atomic activities are used to define basic operation execution request or response messages directed to external services. Complex (or structured) activities are composed of atomic activities and enable the specification of sequential or parallel processing, and condition iterations. In addition to atomic and complex activities, WSCI provides a special type of activity, called *process* that can be used to define a *context* of an execution. A context refers to a scope in which a set of

activities is executed. Therefore the process activity enables modelling a portion of the whole interaction that is labelled with a name and it can be reused inside the WSCI document by referencing its name.

Table 2.3 presents a brief summary of atomic and complex activities of WSCI

Table 2.3: Summary of WSCI activities

Atomic activity	**Description**
Action	This activity is used to specify exchange messages with other services, i.e. receive a message or send a message and wait for a reply.
Call	This activity is used to instantiate a process and wait for its completion.
Spawn	This activity is used to instantiate a process in a parallel thread of control.
Join	This activity waits for a process to complete that has been instantiated using <spawn>
Fault	This activity is used to signal a fault in the context it is in.
Delay	This activity is used to express a time interval, i.e. it introduces a pause in the process.
Empty	This activity is used for no operation purpose. This activity is models the situation where an activity must appear but that activity doesn't show any externally observable behaviour, e.g. internal operation.
Complex activity	**Description**
Foreach	This activity specifies the iterative execution of list of other activities.
until/while	This activity specifies one or more activities that must be executed repeatedly based on the truth value of some condition.
All	This activity includes a set of two or more other activities that should be executed in non-sequential order, i.e. the activities could be executed in any order one by one or possibly in parallel.
Sequence	This activity specifies an ordered list of one or more activities that must be performed in sequential order.
Choice	This activity contains two or more activities and selects one of them based on some event.
Switch	This activity contains a group of activities and selects one of them for execution based on one or more conditions.

2.2.4.2.2 Web Services Choreography Description Language (WS-CDL)

Web Services Choreography Definition Language (WS-CDL) [Wsc05] is an XML based language to describe global or unbiased view of the interactions between two or more services participating in a collaboration to accomplish a common goal. At the time of writing this thesis W3C has released candidate recommendation of WS-CDL [Wsc05] and it has not reached the status of a standard.

In WS-CDL document the choreography of message exchanges between web services is described using three types of activities. These types are, *(i) basic activities (ii) ordering structures* and *(iii) WorkUnits*. Basic activities represent the primitive action to be performed in a choreography, e.g. the exchange of information between participating web services, or manipulation of variable data. Ordering structures define the ordering rules of actions to be

performed in a choreography. These structures enclose basic activities or other ordering structures that should be performed in specific order determined by the semantic of the ordering structure. WorkUnits are used to describe conditional and repeated execution of an activity. A WorkUnit encloses an activity and may be associated with a guard condition and/or a repetition condition. The enclosed activity is executed one or more times based on the evaluation of the guard and/or the repetition condition.

Table 2.4 presents a brief summary of basic activities and ordering structures of WS-CDL

Table 2.4: Summary of WS-CDL basic activities and ordering structures

Basic activity	**Description**
Interaction	This activity is used to specify exchange of information between participants.
Perform	This activity is used to invoke another choreography to be performed within the context of the executing choreography.
Assign	This activity is used to copy data from one variable to another variable.
SilentAction	This activity specifies non-observable behaviour of a participant, i.e. internal operation of a participant.
noAction	This activity specifies a point in the choreography where a participant does not perform any action.
Ordering Structure	**Description**
Sequence	This ordering structure contains an ordered list of other activities or ordering structures that must be performed sequentially in the exact order of their listing.
Parallel	This ordering structure contains one or more activities or ordering structures that are performed in any order or concurrently.
Choice	This ordering structure specifies that only one of two or more activities may be performed based on a Boolean condition or on the occurrence of one among a set of competing events.

2.2.4.3 Comparisons between Service Orchestration and Choreography Standards

In this section we review the characteristics of the web service orchestration and web service choreography languages presented in the previous section and compare the languages in terms of these characteristics. Recently several initiatives have been taken to identify the general characteristics of web service orchestration and web service choreography languages [Pel03a][Pel03b][Tal05][Men04][Mil04][Yus04][Cla05][Sol03]. Here we compare BPEL, OWL-S, WSCI and WS-CDL based on the most essential characteristics found in the literature. These characteristics are described below

Modelling Constructs: A web service orchestration language or a web service choreography language should provide rich collection of modelling constructs to support communication with other web services and handle workflow semantics.

Recursive Composition: A web service orchestration language or a web service choreography language should support recursive composition of work flows. The language should allow to model complex workflow by combining basic and structured activities. In addition it should allow further composition of the workflow with external workflow.

Correlation: A web service orchestration language or a web service choreography language should have mechanism to support stateful workflow. A workflow could have multiple instances at any time, communicating with the same or different services. The language should have a mechanism to deliver messages to the correct instance of the workflow.

Exception Handling: A web service orchestration language or a web service choreography language should have mechanism to generate, catch and handle errors during the execution of a business process.

Non Functional Property Specification: A web service orchestration language or a web service choreography language should support the specification of non functional requirements (e.g. security, dependability or performance requirements) of the composed system.

Semantic Description: A web service orchestration language or a web service choreography language should provide semantic description of the composed process, which allows dynamic service discovery and automated composition of web services.

Formal Semantics: A web service orchestration language or a web service choreography language should be based on some formalism so that the correctness of a composed process can be verified.

Tool Support Available: A web service orchestration language or a web service choreography language should be supported and accepted by wide range of communities including academia and industry.

Table 2.5 compares BPEL, OWL-S, WSCI and WS-CDL in terms of the characteristics defined above. A textual description of the comparison follows the table.

Table 2.5: Comparison of web service orchestration and web service choreography languages

Characteristics	BPEL	OWL-S	WSCI	WS-CDL
Modelling Constructs	high support	high support	High support	High support
Recursive Composition	support	high support	high support	high support
Correlation	high support	No support	high support	support
Exception Handling	high support	No support	High support	High support
Non Functional Property Specification	No support	Low support	No support	No support
Semantic Description	No support	support	No support	No support
Formal Semantic	Indirect support	support	No support	Indirect support
Tool Support Availability	High support	support	Low support	No support

Modelling Constructs: Modelling constructs expose the expressiveness of a language, i.e. how powerful and flexible is the language to model a workflow. All the four languages BPEL, OWL-S, WSCI and WS-CDL provide ample modelling constructs to define a workflow. All these languages provide basic constructs for primitive actions like communicating external web services and structured constructs for flow control like sequential or parallel execution of primitive actions. However OWL-S offers to express preconditions and effects of a web service execution which makes it more expressive than the other three languages.

Recursive Composition: In terms of recursive composition of workflows, OWL-S, WSCI and WS-CDL are ahead of BPEL. In BPEL structured activities allow arbitrary nesting of basic activities or other structured activities, and also BPEL workflow can be exposed as a web service. But BPEL does not allow to use a part of a workflow from other parts of the same workflow, which is allowed in OWL-S, WSCI and WS-CDL.

Correlation: BPEL and WSCI provide explicit constructs for message correlation to synchronise messages received by the workflow from different services. WS-CDL does not have explicit construct to support message correlation, but message synchronisation is achieved using the identity element of a channel. Correlation is currently not supported by OWL-S.

Exception Handling: Exception handling constructs are present in BPEL, WSCI and WS-CDL to catch run time exception, but no such mechanism is offered by OWL-S.

Non Functional Property Specification: None of the languages, other than OWL-S, has any support to specify non functional requirements of the composed system. Only OWL-S offers the specification of very limited non functional properties, namely quality of service.

Semantic Description: OWL-S based on description logic provides semantic description of the composite service which enables automation of service composition. BPEL, WSCI and WS-CDL do not provide any semantic description of the composite process.

Formal Semantic: OWL-S is based on description logic language. Therefore OWL-S process model can be expressed in axiomatic rules of first order logic or Petri Nets [Nar02]. Although BPEL is based on XLANG and WSFL that are rooted in Pi-calculus and Petri Nets respectively, BPEL itself does not have any formal semantics. Recently several efforts have been taken to define formal semantics of BPEL process [Ouy05][Sta04][Far05]. Most of these efforts translated BPEL process into Petri Nets. Similar effort is evident for WS-CDL [Hon06]. WSCI does not have formal semantic support.

Tool Support Availability: As for tool support, BPEL has achieved the widest acceptance both from industry and academia. Most major software vendors including IBM, Microsoft, BEA expressed their support behind BPEL. At the time of writing this thesis OASIS is in the process of standardizing the BPEL specification and they have released a public review draft on August 23, 2006 [Wsb06]. Because of wide acceptance, a number of tools are now available that support design and/or execution of BPEL process, including BPWS4J from IBM [Bpw03], Oracle BPEL Process Manager from Oracle [Ora04], Biztalk Server from Microsoft [Mic04]. OWL-S mainly gained support from academic world [Yus04] and there are few prototypes available that support design in OWL-S. But no tool available to execute OWL-S process. Similarly very little tool support is available for WSCI. SunONE WSCI Generator releases by Sun Microsystems supports WSCI [Sun03]. Currently no tool support is available for WS-CDL.

2.2.5 Specification and Management of Service Level Agreements

With the maturing of web-service technologies and standards, the importance of being able to specify and monitor agreements between providers and consumers of web-services setting the objectives that the services should satisfy and the penalties that may arise when they fail to do so is widely recognised in industry and academia [Jin02][Men02][Dav02][Tia03][Tia04][Aie05]. As a result of this recognition, there have

been proposals for standardised ways of specifying such agreements, including WSLA [Kel02, Lud03], WS-Agreement [And04] and WS-Policy [Sch04]. In this section we present an overview of these standards.

2.2.5.1 Web Service Level Agreement (WSLA)

The Web Service Level Agreement (WSLA) framework is an XML based language for specifying service level agreements between a service provider and service consumer and the obligations of the parties involved [Kel02, Lud03]. WSLA supports the specification of service level agreements using:

(i) *Resource Metrics* are the basic parameters to be observed and are retrieved directly from the resources (e.g. routers, servers, instrumented applications), usually managed by the service provider. For example a resource metric could be a counter that counts the number of invocations made to an operation in a service. To retrieve a resource metric a *measurement directive* should be specified in WSLA document. Measurement directive contains the command and context information needed to access the resource metric.

(ii) *Composite Metrics* are computed by combining resource metrics, where a function explains how to compute a composite metrics (e.g. average, sum), either from resource metrics or composite metrics. For example if a resource metric counts the number of invocations made to an operation in a service, a composite metric to measure the average number of invocations made to that operation in a given period can be defined based on this resource metric.

(iii) *Service Level Agreement (SLA) Parameters* enable a specific service consumer to add information to the specification of a metric in order to enable its evaluation (e.g. source of the metric, high/low value of the metric). For example, if a composite metric counts the average number of invocation made to an operation in a service for a given period, an SLA parameter can specify range (i.e. max value and min value) of this composite metric for a specific consumer of the service

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:wsla="http://www.ibm.com/wsla"
   targetNamespace="http://www.ibm.com/wsla" elementFormDefault="qualified">

  <!-- Global WSLA structure -->
  <xsd:complexType name="WSLAType">
    <xsd:sequence>
      <xsd:element ref="wsla:Parties"/>
      <xsd:element name="ServiceDefinition"
         type="wsla:ServiceDefinitionType" maxOccurs="unbounded"/>
      <xsd:element name="Obligations" type="wsla:ObligationsType"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>

  <xsd:element name="SLA" type="wsla:WSLAType"/>

  <!-- Party Definitions -->
   .... ... ...
    <xsd:complexType name="PartiesType">
      <xsd:sequence>
        <xsd:element name="ServiceProvider" type="wsla:SignatoryPartyType"/>
        <xsd:element name="ServiceConsumer" type="wsla:SignatoryPartyType"/>
        <xsd:element name="SupportingParty" type="wsla:SupportingPartyType"
                        minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="Parties" type="wsla:PartiesType"/>

    <!-- Service Definitions -->

    <xsd:complexType name="ServiceDefinitionType">
      <xsd:complexContent>
        <xsd:extension base="wsla:ServiceObjectType">
          <xsd:sequence>
            <xsd:element ref="wsla:Operation" minOccurs="0"
                                                maxOccurs="unbounded"/>
            <xsd:element ref="wsla:OperationGroup" minOccurs="0"
                                              maxOccurs="unbounded"/>
            <xsd:element ref="wsla:WebHosting" minOccurs="0"
                                                maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
      </xsd:complexType>
      ... ... ...
    <!-- Guarantees -->
      ... ... ...
    <xsd:complexType name="ObligationsType">
      <xsd:complexContent>
        <xsd:extension base="wsla:ObligationObjectType">
          <xsd:sequence>
            <xsd:element ref="wsla:ObligationGroup" minOccurs="0"
                                        maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
      ... ... ...
</xsd:schema>
```

Figure 2.12: Overview of WSLA schema

(iv) *Business Metrics* provide the basis of customer's risk management strategy by adding customer specific financial terms to SLA parameters. For example business metric can associate penalties or bonuses to SLA parameters.

The general scenario assumed by the WSLA framework is that a service provider exposes a set of resource metrics and composite metrics, and a service consumer defines SLA parameters and business metrics. A WSLA is agreed and signed by both parties (known as *signatory parties* [Kel02, Lud03]) through negotiation. Signatory parties may monitor the WSLA or they may wish to employ one or more third parties (known as *supporting parties*[Kel02, Lud03]) to monitor the WSLA.

A detail description of the WSLA schema is beyond the scope of this thesis. Figure 2.12 shows an overview of the WSLA schema. According to the WSLA XML schema, a WSLA document is divided into three sections. The parties (e.g. signatory party, supporting party) involved in an agreement are introduced in the *Parties* section. In the *Service Description* section the characteristic of the service and its observable parameters (e.g. resource metrics, composite metrics and SLA parameters) are defined. In the *Obligations* section, various guarantees and constraints imposed on the SLA parameters are described.

WSLA language and its associated architecture are generic enough to cover a wide range of service level agreements. Also the extensible mechanism offered by WSLA allows domain specific language specification. However WSLA does not provide the specification of functional requirements for web services and WSLA may prove relatively complex to service consumer as a WSLA document contains a level detail that is irrelevant to service consumer. For example, what resources are used to make measurement, or what is the mechanism used for measurement, these are in most cases concerns of the service provider.

A prototype of WSLA framework has been released by IBM that publicly available from IBM web site [Kel03].

2.2.5.2 Web Services Agreement (WS-Agreement)

WS-Agreement [And04] is a specification that defines a protocol to establish agreements between service providers and service consumers, an XML based language to specify

agreements and a runtime agreement monitoring interface to monitor the compliance of the offering of a service with the agreement at runtime.

A WS-Agreement is an XML based language for describing functional and non-functional properties of a service, e.g. guarantees over service level objectives as well as conditions, which must exist for the service level objective to be fulfilled. It is also possible to express business values associated with these objectives, rewards for the fulfilment of the objectives, and penalties for failure to fulfil an objective.

Figure 2.13 shows an overview of the WS-Agreement schema.

According to WS-Agreement a web-service agreement specification has two sections, namely the *context* and the *terms* section. The *Context* section contains various meta-data about an agreement, e.g. the duration of the agreement and links to other agreements which are related to this agreement. The Context section also contains the description of the parties involved in the agreement. The *Terms* section contains the elements that describe the agreement itself. This section is divided into two sub sections, namely the *Service Description Terms* and *Guarantee Terms*. Service description terms constitute the basic building block of an agreement and define the service functionalities to be delivered under the agreement. An agreement may contain any number of service description terms. A guarantee term specifies an assurance on service quality associated with the service described by the service description terms. Both the service description terms and the guarantee terms can be composed using the compositors of the WS-Policy specification [Sch04, Lud04] (see Section 2.2.5.3 for an overview).

In the life cycle of creating, agreeing, monitoring and applying an agreement, an agreement initiator creates and sends an agreement template to the consumer. An agreement template is defined by adding a new section, namely *Creation Constraints*, to the agreement structure described above. This new section contains constraints on possible values of terms for creating an agreement, i.e. constraints specify the valid ranges or distinct values that the terms may take in the agreement. The consumer fills in the template and sends it back to the initiator as an offer. The initiator notifies the consumer of the acceptance or rejection of the agreement depending on the availability of resources, the service cost etc. The agreement is monitored when at least one service involved in the agreement is running.

```
<xs:schema targetNamespace="http://www.ggf.org/namespaces/ws-agreement"
   xmlns:wsag="http://www.ggf.org/namespaces/ws-agreement"
   xmlns:wsbf="http://www.ibm.com/xmlns/stdwip/webservices/WS-BaseFaults"
   xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified" attributeFormDefault="qualified">

      ... ... ...
   <xs:element name="Template" type="wsag:AgreementTemplateType"/>
   <xs:element name="AgreementOffer" type="wsag:AgreementType"/>
   <xs:element name="Name" type="xs:NCName"/>
   <xs:element name="Context" type="wsag:AgreementContextType"/>
   <xs:element name="Terms" type="wsag:TermCompositorType"/>

   <xs:complexType name="AgreementContextType">
     <xs:sequence>
       <xs:element name="AgreementInitiator" type="xs:anyType"
                          minOccurs="0"/>
       <xs:element name="AgreementProvider" type="xs:anyType" minOccurs="0"/>
       <xs:element name="AgreementInitiatorIsServiceConsumer"
                 type="xs:boolean" default="true" minOccurs="0"/>
       <xs:element name="TerminationTime" type="xs:dateTime" minOccurs="0"/>
       <xs:element name="RelatedAgreements"
          type="wsag:RelatedAgreementListType" minOccurs="0"/>
       <xs:any namespace="##other" processContents="lax" minOccurs="0"
                                      maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:anyAttribute namespace="##other"/>
   </xs:complexType>

   <xs:complexType name="TermCompositorType">
      <xs:sequence>
       <xs:choice maxOccurs="unbounded">
         <xs:element name="ExactlyOne" type="wsag:TermCompositorType"/>
         <xs:element name="OneOrMore" type="wsag:TermCompositorType"/>
         <xs:element name="All" type="wsag:TermCompositorType"/>
         <xs:element ref="wsag:ServiceDescriptionTerm" minOccurs="0"/>
         <xs:element ref="wsag:GuaranteeTerm" minOccurs="0"/>
       </xs:choice>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="AgreementTemplateType">
     <xs:sequence>
       <xs:element ref="wsag:Name" minOccurs="0"/>
       <xs:element ref="wsag:Context"/>
       <xs:element ref="wsag:Terms"/>
       <xs:element name="CreationConstraints"
                 type="wsag:ConstraintSectionType"  minOccurs="0"/>
     </xs:sequence>
   </xs:complexType>

   <xs:complexType name="AgreementType">
     <xs:sequence>
       <xs:element ref="wsag:Name" minOccurs="0"/>
       <xs:element ref="wsag:Context"/>
       <xs:element ref="wsag:Terms"/>
      </xs:sequence>
   </xs:complexType>
      ... ... ...
</xs:schema>
```

Figure 2.13: Overview of WS-Agreement schema

The WS-agreement specification offers a rich language for capturing and presenting real world assurances and requirements of web services in terms of service level objectives (SLOs), qualifying conditions and business values. But the specification does not specify any domain-specific terms describing the service level objectives, rather the framework leaves it open to its users to decide what assertion language should be used to specify the service objectives. For example, both the service description properties and the guarantee terms can be composed using the compositors of the WS-Policy specification. Although this option allows users to use existing languages to describe particular aspect of a service, it puts the users in a situation to be able to deal with a variety of specifications. The negotiation protocol as specified in the WS-Agreement Specification is only a two step conversation, in which the service consumer receives either an accept or a reject message from the service provider in response to an agreement request. In real world scenario this is a very limited negotiation model, where an iterative negotiation would be more feasible.

2.2.5.3 Web Services Policy Framework (WS-Policy)

WS-Policy specification defines a syntax and semantic that enables web service providers to advertise their policies and the web service consumers to specify their policy requirements for a web service [Sch04]. In W3C glossary a policy is defined as a constraint on the behaviour of a web service [Wsg04]. WS-Policy framework provides a means of documenting a broad range of characteristic aspects, like security, QoS characteristics or transactionality, of a web service as policies.

WS-Policy is an XML based language that can be used to express these aspects. According to the schema a policy is known as *policy expression*, which is a collection of *policy assertions*. A *policy assertion* is the core element that can be used to define individual preferences, requirements, capability or other characteristics of a web service based system. For example some assertions may specify requirements that manifest on the wire, e.g. authentication scheme or transport protocol selection, again some assertions may specify requirements that are critical to proper service selection and usage, e.g. privacy policy or QoS characteristic. Extensibility mechanism of WS-Policy allows that policy assertions can be defined by another specification. For instance, Web Service Policy Assertions Language (*WS-PolicyAssertions* [Nad03]) specification defines a set of general message assertions and the Web Service Security Policy Language (WS-SecurityPolicy [Nad05]) specification defines a set of common security related assertions. In policy expressions, assertions can be combined

by using *policy operators*, which essentially give the logical AND, OR and XOR of policy assertions. A policy expression can be bound to an entity (e.g. a web service endpoint, object or resource), which is known as *policy subject.*

But WS-Policy framework does not define how policies can be attached to a policy subject and leave this issue to other specifications. The Web Service Policy Attachment (*WS-PolicyAttachment* [Sha04]) specification is one such specification that defines how to attach policy expression to XML elements, WSDL definitions and UDDI entities.

2.3 Runtime Monitoring of Software

2.3.1 A General Framework for Runtime Monitoring

Software monitoring is the activity of determining at runtime whether a software system is operating according to requirements set for it during its development. Figure 2.14 presents a general framework for the overall process of software requirements monitoring that we use in this thesis to provide a comparative view of different approaches found in the literature. In this framework, requirements are expressed in some specification language. Software source code is developed according to the requirement specification. Monitoring events are identified from the requirements specification by applying different approaches, e.g. goal driven requirements acquisition [Dar93]. Monitorable events are expressed in some language. The monitor is developed to trace monitorable events (typically sequences of such events) at runtime. Various mechanisms are adopted to generate runtime events, such as (i) built in tools (e.g. various management facilities offered in a distributed system [Sam92]); (ii) Code instrumentation i.e. insertion of informative statements into the source code to generate desired events at runtime. Instrumentation is driven by a specification that is derived from the monitorable event specification following composition rules of the instrumentation tool. The instrumentation tool instruments the source code based on this specification. (iii) *Architectural reflection* is another option to generate monitoring events. A reflective software system is a system that performs computation about its own software architecture, such as memory consumption etc. Hence a reflective system maintains a runtime description of the overall architecture and all the states of the system conveying information relevant to operate on the architecture.

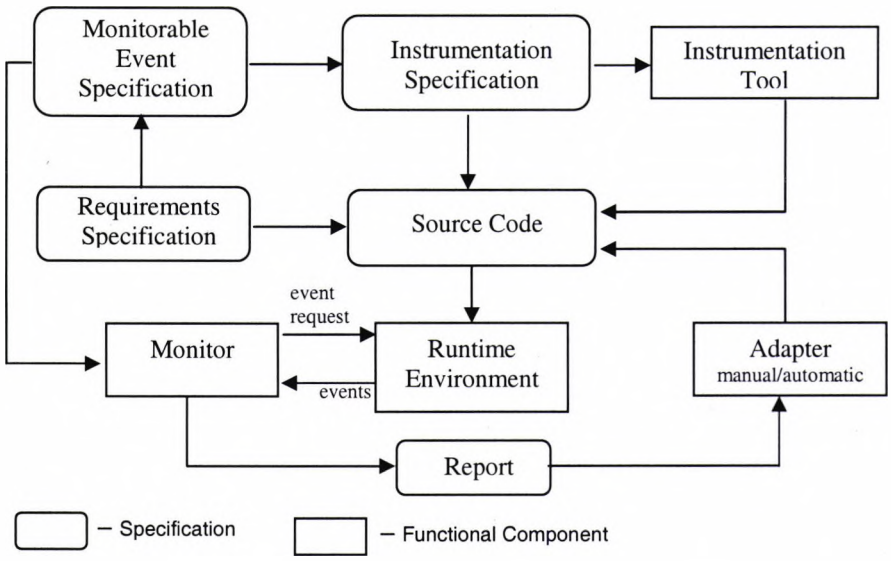


Figure 2.14: A general framework for requirements monitoring

At runtime the monitor receives monitorable events from the source code or the instrumented source code, checks the event sequence against the requirements specification and generates a report if a violation is detected. The violation report may be used, either manually or automatically, to tune the software to reconcile its runtime behaviour with requirements.

2.3.2 Strands of Research in Software Monitoring

In the following sub sections we portray the current trend of research in the requirements monitoring of legacy software as well as web service based systems. At the end of each sub section we present a comparative summary of the relevant technique that conform to the general framework for requirements monitoring shown in Figure 2.14 and for which implementation detail is found in the literature. The comparative summaries of the works are based on different implementation artefacts of the general monitoring framework. These artefacts are,

Requirement specification language: Requirement specification language is the language used in the work to express the requirements set for the system to be monitored.

Monitorable event identification process: Event Identification process specifies the mechanism used in the work to identify the events needed to monitor the system and that can be captured or generated during the execution of the system.

Event specification language: Event specification language is the language used in the work to specify the monitorable event patterns.

Runtime event generation mechanism: Event generation mechanism is the mechanism used in the work to generate or capture runtime event to monitor the system.
Adaptability: Adaptability specifies the mechanism used in the work to reconcile the runtime behaviour of the monitored system with its requirement, in case the requirement is violated.

2.3.2.1 Requirements Monitoring Using FLEA

Fickas and Feather suggest the necessity and objectives of requirements monitoring and the benefits that different stakeholders can gain thereof [Fea95]. They focus on requirements monitoring in dynamic environments, i.e. environments in which a system may change over time. In such an environment assumptions about the environment of a system at design time may prove invalid when the system is in operation. This makes necessary the development of mechanisms to support the monitoring and evolution of the system. The most critical capabilities that system should have in order to be able to respond to changes in their environments are (i) to detect the exact point when the system needs to be evolved, i.e. the point in time when the design time assumptions about the environment become invalid, and (ii) to adjust the system in order to maintain their requirements specification at that point. Fickas and Feather argue that the identification of evolution points in the lifetime of a system can be identified only by monitoring the system and its runtime environment and present a general approach for monitoring requirements in a dynamic environment. A key element of this approach is to establish relationships amongst:

- The overall requirements of the system
- The assumptions made about the current state of the environment of the system. These assumptions form the context in which requirements are formulated. and
- The set of remedial evolutions available when mismatches develop between assumptions and the current environment.

A close analysis of the relationships among the requirements and assumptions can help to find out what to monitor. This results in the creation of a monitoring specification. From this monitoring specification actual runtime monitoring code can be created by applying some existing tools and techniques. A runtime requirements monitoring architecture called AMOS (Assumptions MOnitoring System) has been proposed in [Coh97] following this approach. A runtime requirements monitoring environment should allow the analyst to specify a wide range of user's requirements and assumptions. Specification of requirements and assumptions should be compiled automatically into runtime monitoring code and the framework should be readily applicable to systems that have not been designed with an intention to be monitored.

AMOS is proposed to meet these requisites to make it supportive for runtime requirements monitoring. AMOS uses the language FLEA (Formal Language for Expressing Assumptions) [Fea97] to specify requirements and assumptions that can be monitored at runtime. FLEA is an event specification language with a small but powerful set of constructs to capture wide range of monitoring properties (e.g. performance, safety properties). A compiler converts the FLEA specification into runtime monitoring code using an active database (i.e. a database with triggers). The monitor observes the interactions between the monitored system and the environment using a generic data-gathering mechanism, (e.g. message-bus) and reports all the violation of requirements and assumptions. In [Fea98] Feather et al move one step forward to deal the second issue raised in [Fea95], that is the adjustment of a system following the identification of the need for evolution. In that paper the authors present an architecture for a self-adaptive system that can reconcile its runtime behaviour with its requirements. To achieve the self-adaptability, they propose an architecture of cooperating software agents that has alternative system designs represented as system parameters and/or alternative refinement trees. A system can adapt itself by either tuning some parameter or switching to an appropriate design at runtime whenever some violation occurs. This approach assumes two main stages:

(i) At development time, requirements are initially specified as goals and are subsequently refined into assertions based on goal-driven requirements elaboration techniques [Dar96]. All the monitorable and controllable parameters are identified at this stage. Appropriate reconciliation tactics (i.e. parameter tuning or alternative system design) are also identified and associated with each breakable assertion. Breakable assertions are translated into FLEA events.

(ii) At runtime, a monitor captures events from each software agent in the architecture through an appropriate communication channel and produces a violation file. In case of passive communication, the monitor polls each software agent at specific time interval and in active communication every software agent notifies events to the monitor when their monitored parameters change. A customizer analyses the violation file and applies the reconciliation tactic specified for the violation at the development level. Whenever the system shifts to an alternative design, the monitor needs to be reconfigured to retain the consistency between the monitoring event sequence and the breakable assertions in the current specification.

Summary[1]

Technique	Requirements Monitoring in Dynamic Environment [Fea95]
Requirement Specification Language Used	Natural Language
Monitorable Event Identification Process	Goal Directed Requirement Acquisition
Monitorable Event Specification Language Used	-
Runtime Event Generation Mechanism	Built in event monitoring tool, e.g. user management facility provided in licence manager software.
Adaptability	Identify the reconfiguration point of a software in dynamic environment and adjust it manually
Comments	+ refinement of requirements to sub-requirements and identification of possible remedial actions in case of violation of those sub-requirements enables to design a self adaptive system - Manual adjustment of the monitored system if a violation is detected.
Technique	AMOS [Coh97]
Requirement Specification Language Used	-
Monitorable Event Identification Process	Events are provided by user
Monitorable Event Specification Language Used	FLEA
Runtime Event Generation Mechanism	Monitor observes the messages passed between the monitored system and its environment
Adaptability	An action for each monitoring condition is defined in a database. When the condition is satisfied at runtime corresponding action is executed
Comments	+ allows to express a wide range of users' requirements and assumptions, also permits to add monitoring queries during the execution time of the monitor - Requirements /assumptions must be expressed in terms of monitoring events. Therefore user must know runtime events beforehand.
Technique	Reconciliation of requirements and system behaviour [Fea98]
Requirement Specification Language Used	KAOS assertions
Monitorable Event Identification Process	Formal refinement pattern for goal directed requirements elaboration
Monitorable Event Specification Language Used	FLEA
Runtime Event	Monitor communicates to each software agent through an appropriate channel, such as active or

[1] In all the summary tables in this chapter, in the comments row a (+) sign stands for positive feature and (–) stands for negative feature.

Generation Mechanism	passive communication
Adaptability	Self-adaptability of a system in a dynamic environment is achieved by providing alternative system designs. System switches to apposite design at runtime when some violation occurs
Comments	+ Automatic recovery of the monitored system based on the detected deviations.

2.3.2.2 Requirements Monitoring Using Instrumented Code

Robinson [Rob02] exploits the concept of instrumentation and proposes a framework to continually analyse requirements during the runtime execution of software. Since requirements monitoring may prove expensive the author suggests monitoring of 'suspect requirements'. Suspect requirements can be identified as weakened conditions of requirements by applying the rules described in the goal-driven requirements elaboration method of [Dar96]. The major steps in Robinson's approach, which is essentially a refinement of the model described in [Fea98], are:

(i) Initially high level requirements are expressed in some formal specification language e.g. KAOS [Dar93].

(ii) An analysis and design model is subsequently constructed using some modelling language e.g. UML [Uml03]. This model facilitates traceability between the requirements and the software.

(iii) An implementation of the design model developed in (ii) is constructed. This implementation should maintain static traceability of definitions and dynamic traceability of class instances at runtime. Some informative statements should be inserted into the software code to generate a stream of information interpretable by the monitor, which is known as instrumentation. The framework uses the instrumentation tool JOIE [Coh98] to instrument Java class files.

(iv) At runtime the monitor continually views the stream of information produced by the instrumented code. It uses assertion checking to monitor specific suspect condition and produces warnings if some requirements are about to be violated. Model checking is used to determine if a requirement could fail in a future state. A tool (monitorLog2Java) is used to translate the monitored program event log into a Java programme that contains only the sequence of method calls in the original Java programme. Another tool (java2spin) generates a state-based model (Promela model) from this modified java programme, which is checked for failure of suspect conditions and requirements using the spin model checker [Hol97].

Dingwall-Smith and Finkelstein describe an architecture for runtime monitoring of system goals as part of normal system operation [Din02]. High level goals are decomposed by applying the KAOS approach to develop the monitor. The architecture uses Hyper/J [Hyp03] to instrument the class files of the software to be monitored. Hyper/J is a tool that allows the extraction of concerns from some existing software (written in Java) and the merging of these concerns with some new concerns to develop a new software. According to the Hyper/J specification, a concern is a coherent area of interest in the body of a software system, including class, aspect or feature. Instrumentation class files are inserted into the class files of the software system to be monitored following Hyper/J composition rules. At regular intervals, the monitoring system reads the events generated by the instrumented code and determines whether the monitored goal is satisfied. The approach has been demonstrated using the Limewire Gnutella [Gnu] peer to peer file sharing system as test bed.

Summary

Technique	Requirements Monitoring using instrumentation [Rob02]
Requirement Specification Language Used	KAOS assertion
Monitorable Event Identification Process	UML is used as intermediate language to maintain traceability between high level requirements and the software.
Monitorable Event Specification Language Used	
Runtime Event Generation Mechanism	Code Instrumentation.
Adaptability	
Comments	+ Monitoring of suspect conditions enables to raise increasingly urgent notifications prior to the failure of a requirement. This could be useful in taking remedial action in real time. - Monitored software is instrumented to generate runtime events that can degrade the performance of the software.
Technique	Requirements Monitoring by way of Aspects [Din02]
Requirement Specification Language Used	KAOS assertions
Monitorable Event Identification Process	Goal-directed requirements acquisition [Dar93]
Monitorable Event Specification Language Used	High level requirements expressed in temporal language are hard coded to the monitor to maintain traceability. No intermediate event definition language is used
Runtime Event Generation Mechanism	Code instrumentation by adding aspects using through Hyper/J composition rules.

Adaptability	-
Comments	- Monitored software is instrumented to generate runtime events that can degrade the performance of the software.

2.3.2.3 Runtime Requirements Engineering & Monitoring of Personal and Ephemeral Requirements

Fickas et al [Fic02c] focus on the cost effectiveness of formal analysis of requirements. In traditional formal analysis, a software model or a property is changed to eliminate a detected requirement violation. This approach may prove expensive and the cost involved in this approach may not worth the benefit it produces. The authors are interested in an untraditional approach; that is to carry on the requirements analysis at run time, i.e. what refers to "runtime requirements engineering", without taking any corrective measure at the static analysis phase. They establish their argument by providing an example of a fault protection engine (FPE) of a spacecraft. The FPE can diagnose and treat runtime faults of the spacecraft by running repair routines on request of the environment e.g. a physical sensor or a ground controller. Fault repair requests are placed in a queue and then cleared after a repair routine for each request has been executed. FPE is expected to run a repair routine for every fault detected. This requirement is translated into a Promela automaton which is then checked using the spin [Hol97] model checker. The authors come to the decision that erroneous engineering assumptions about the runtime environment may result a non-trivial number of causes for system failure during analysis phase and most of these causes can be explained without taking any remedial action. So they suggest to make assumptions at analysis, but record and carry them to runtime for monitoring and use the monitoring information in different ways, including (i) the generation of warning that failures are certain in the next few states of the system, (ii) controlling the environment of the system in some way (iii) modifying the component that has created the violation in order to get rid of the violation without relying on help from the environment.

Fickas and Hall present various issues to deal higher level requirements placed by diverge users in a multi-stakeholder distributed system (MSDS) like Internet email system, networks of Web services, modern telephone network and the Internet itself [Fic02d]. Open systems like these are dynamic and have no regular shape, comprising whatever components are available at a particular instance of time. Most of the time these components are beyond user's control and cause the high level requirements to fail in various ways. Based on the characteristics of open systems the authors introduce two new types of requirements, (a) ephemeral requirements – these are requirements that have a fixed life span and may occur

one or more times; (b) personal requirements – these are requirements which are very much specific to single user rather than to all users of the system. An example of a personal requirement for an email system is the sender of an email should receive a reply from the receiver of the email. Clearly this requirement is personal to sender only and it is also ephemeral as the sender needs only a single response from the receiver. In [Fic02a] personal requirements has been treated by Fickas and others. The project attempts to develop a system to deliver software tools to those with traumatic brain injury (TBI). One has to consider three issues in developing such system: (a) the requirements of each individual should be identified (b) a system should be developed and deployed to meet those requirements and (c) the deployed system should be monitored at runtime to watch if it is behaving as it was planned. To handle the first issue a personal requirements engineering process, called CORE (Comprehensive Overview of Requisite Email skills) has been introduced assuming email system will be the application to be deployed. CORE process enables evaluation of a TBI survivor in terms of his goals, corresponding skills and his/her environment. This information eases the development and delivery of the software tool to meet the requirements. A case study of runtime requirements monitoring of a deployed email system for TBI survivors in a composite system is presented in [Fic02b]. User requirements are refined according to the KAOS guidelines and then restated using Timeline Editor (a tool for specifying time-based events). The outputs from the timeline editor are a Buchi-automaton shown graphically and the same automaton in the Promela [Hol91] never-claim form. An event-monitoring tool, called Emu, is used to perform the monitoring. The automaton in the Promela form obtained from timeline editor is translated into an Emu event tree by adding context information (e.g. timing). Emu accepts the requirements specification in the form of an event-tree as input and receives events from clients at runtime through a distributed listener architecture. Whenever a received event from a client matches a trigger event of the input tree, Emu creates an internal representation of the event tree and starts monitoring the rest of the events in the tree. A runtime gauge shows the status of the requirements being monitored. This status can be used in various ways. For example, manual or automatic repair procedures can be invoked when a violation is detected.

Summary

Technique	Handling of Personal and Ephemeral requirements [Fic02a] [Fic02b] [Fic02c]
Requirement Specification Language Used	Natural Language
Monitorable Event Identification Process	Goal-directed requirements acquisition

Monitorable Event Specification Language Used	Promela automata [Bar04b]
Runtime Event Generation Mechanism	Monitoring tool (Emu) receives events from the clients through a distributed listener architecture. Clients generates events using built in event generation tools.
Adaptability	-
Comments	- The approach can not monitor liveness properties. Liveness properties are translated to safety property by applying time constraints which may result in detection of incorrect requirements violation.

2.3.2.4 Requirements Monitoring Using Reflection

Finkelstein and Savigni present a framework that facilitates implementation of context-aware services [Fin01]. Traditional software lacks ability to adapt itself to a changing context or changing requirements. The framework that they propose is meant to address this shortcoming. This framework is based on a reflective approach and uses a meta-level description of the actual service implementation. The meta-level service description maintains a runtime representation of system behaviour that reifies the actual system behaviour in the sense that changes in the actual system behaviour are reflected in the meta-level description. Requirements are defined as possible ways of achieving high level goals. A real time description of the environment, known as context, can cause requirements to change due to environmental changes and thereby accomplish the high level goal. For instance, in an m-commerce service available low bandwidth would change a requirement of presenting a high quality image to a requirement of presenting the same image in degraded quality. The environment can also influence the service to change it in unpredictable way that forces a change in meta-level description of the service. So a runtime violation can be identified by comparing the meta-level description against the requirements specification.

Efstratiou et al [Efs02] argue about the need for system wide co-ordinated response of adaptive applications to avoid conflicts or sub-optimal performance. Adaptive applications running in the same context may have individual adaptive mechanism that may lead to unwanted conflicting side effects. For example one application may reduce the use of network to save power consumption, while another application running in the same host may try to increase the use of network as it is released by the first application. The authors propose a platform to handle situations like this. In their platform, adaptive mechanisms of individual applications are co-ordinated by a single adaptation controller. Applications in a system expose their adaptive mechanisms and set of state variables to the adaptation controller and also at runtime the applications send events to the adaptation controller. These events reflect

the change in value of the state variables. The adaptation controller analyses the change in state variables and invokes appropriate adaptation mechanism according to some predefined policy rule. A policy language has been devised based on event calculus to define the policy rules. A policy rule consists of two parts. The first part specifies a set of conditions based on the state variables and second part specifies a set of actions that should be performed if the condition is satisfied.

Clarke and Osterweil suggest an approach leading to self-evaluating and self-improving software [Cla00]. Human centric software improvement techniques can be automated if the software itself carries out continuous testing, analysis and evaluation. The two complementary evaluation approaches, dynamic testing and static analysis, should go on synergetically to enhance the overall process. The authors argue for the need for self-modifying software systems instead of self-modifying code as the latter is generally difficult or impossible to analyse. They view a software system as a collection of different software artefacts such as requirements specification, architecture, low level design specification, code, test cases, analysis result. In addition to the artefacts software system also includes, a process, known as the modification process, which is responsible for modifying other components e.g. the code. A software system should also have a set of constraints specifying the way in which the different components of the system should be related to each other. The constraints are responsible for identifying the modification point. For example, when test results are inconsistent with the requirements a product modification signal should be raised. In classic software development process product code is isolated from the other components (e.g. testing tool, constraints etc.) at the deployment time that complicates further modification of the code substantially. So the authors propose to deploy product code integrated with the other components to support perpetual testing, evaluation and improvement.

Summary

Technique	Utilising the event calculus for policy driven adaptation on mobile systems [Efs02]
Requirement Specification Language Used	A policy language based on Event Calculus has been devised
Monitorable Event Identification Process	State variables in the application are used as events
Monitorable Event Specification Language Used	Same as the requirement specification language
Runtime Event Generation	Applications send events to the adaptation controller

Mechanism	
Adaptability	Specific adaptive actions are invoked by the adaptation controller based on policy rules
Comments	+ use of event calculus as the base makes the policy definition language more flexible over existing languages to define time relations and conditions - Only non functional requirements are monitored - applications are forced to run under a controller, thereby lose their autonomy.

2.3.2.5 Requirements Monitoring for Safety Critical Systems

Peters presents an approach for automating the generation of monitors from requirements documentation [Pet97]. This approach focuses on requirements specified for safety- or mission-critical real time systems. The author is interested in generating software monitors from system requirements specifications, which are expressed in terms of environmental quantities that can be monitored and controlled by the system. Such monitors can be used to check whether the software behaviour is consistent with the specification. The proposed approach expresses system requirements as a finite state automaton that models the acceptable system behaviour. A state in the automaton represents the system behaviour expressed in terms of predicates (conditions) that characterise some aspect of the monitorable and controllable quantities and each transition is labelled by a predicate that represents a set of events that have similar interpretation with respect to the specification. An implementation of the proposed framework has been presented in [Pet02]. Expected behaviour of the system is stated in tabular relational notations [Par92] using monitorable and controllable quantities. At runtime, the monitor receives environmental quantities (both monitorable and controllable) through some input devices or directly from the system being monitored. Occurrences of events are assumed if some quantity changes its value from the previous state. In such cases the monitor verifies if the transition implied by the changes is acceptable at the particular time according to the requirements specification.

Lutz and Mikulski summarise the evolution of new requirements to the onboard software on spacecraft after launch [Lut01]. The authors consult an institutional database of anomaly reports from three spacecrafts that record the aberrant behaviour, probable reason behind the deviation and possible corrective measure. Most of these anomalies demand modification to the flight software and a close analysis reveals that post-launch requirements changes do not result from previous concerned requirements, but due to hardware failures and rare events. Hardware failures and rare environmental events (e.g. use of obsolete data) prompt the software changes to bridge the gap between the requirements specification and implementation. The authors raise an open research issue – "to what extent it might be

possible, via monitoring, to anticipate some of the rare events or hardware failures that triggered the critical requirements changes on the spacecraft".

Summary

Technique	Requirements Monitoring of Real Time Systems [Pet97] [Pet02]
Requirement Specification Language Used	Tabular Relational notation [Par92]
Monitorable Event Identification Process	Requirements expressed in terms of monitorable and controllable quantities. So events are known beforehand
Monitorable Event Specification Language Used	-
Runtime Event Generation Mechanism	Events are detected by comparing values of monitorable and controllable quantities in successive system states
Adaptability	-
Comments	+ Given a requirements specification document, some modules of the monitor are generated automatically. - offline monitoring. Monitor analyses recorded event traces of the monitored system.

2.3.2.6 Runtime Verification of Java programs

Brorkens and Moller [Bro02a, Bro02b] have developed a tool called jassda that enables runtime checking of Java programs against a Communicating Sequential Processes (CSP) [Hoa85] like specifications. The variant of CSP used in the work, known as CSP_{jassda}, is a specification language that can be used to specify the trace of all possible events emitted by a program during its execution. The jassda framework is based on JDI (Java Debugger Interface), which provides a rich set of functionality to monitor and manipulate an executing Java program in addition to set breakpoints and watch variables. The core component of the framework, known as *Broker*, works as the intermediary between the JDI and the *jassda modules* that will process the events of interest. At the initial phase the broker determines the set of all possible events of the loaded classes and invokes the modules to submit its events of interest. At runtime the broker receives the targeted events and distributes to the respective modules for further processing. Current implementation of jassda provides two modules. The logger module simply writes the received events into a file, which can be used for analysing the program. The trace-checker module analyses the received events on the fly, e.g. the event sequence can be verified against CSP specification.

Kim et al [Kan00, Kim01a, Kim01b] describe a prototype, Java-Mac that monitors and checks a Java program based on a formal specification of systems requirements. The Java-Mac architecture provides two languages.

(i) Primitive Event Definition Language (PEDL) that allows the definition of implementation dependent low-level behaviours, i.e. the events and conditions of the target program. In the current implementation PEDL can be used to define events/conditions using only the primitive data types (both local and global variable) and beginnings/endings of methods in a Java program.

(ii) Meta Event Definition Language (MEDL) which is used to define high level behaviours of the target program using the primitive events and conditions defined in PEDL specification. The architecture works in two phases.

In static phase a PEDL compiler compiles the PEDL specification to generate instrumentation information, which is used by an instrumentor to instrument the target program. The PEDL compiler also generates an abstract syntax tree, which is evaluated by an event recogniser. An MEDL compiler compiles the MEDL spec to generate an abstract syntax tree. At runtime instrumented target program generates events and supplies the events to the event recogniser. The event recogniser in turn sends the events to a runtime checker. The runtime checker evaluates the abstract syntax tree generated by the MEDL compiler against the events. If the runtime checker detects a violation of a property it raises a signal.

Summary

Technique	Jassda [Bro02a, Bro02b]
Requirement Specification Language Used	A variant of CSP, known as CSP_{jassda}
Monitorable Event Identification Process	
Monitorable Event Specification Language Used	Same as the requirements specification language
Runtime Event Generation Mechanism	Java Debugger Interface is used to generate runtime events
Adaptability	-
Comments	+ the requirement specification in CSP_{jassda} is independent of source code, i.e. the approach is applicable even to third party code. - use of JDI to generate runtime event imposes substantial overhead on program execution - use of instrumentation (e.g. in case of method return value) also degrades performance
Technique	Java-Mac [Kan00] [Kim01]
Requirement	MEDL

Specification Language Used	
Monitorable Event Identification Process	Change of values of primitive Java variables and beginnings/endings of method execution are used as events.
Monitorable Event Specification Language Used	PEDL
Runtime Event Generation Mechanism	Instrumentation
Adaptability	Not supported
Comments	+ separation of low level behaviour and high level behaviour language makes the approach applicable to broad range of target platform - Instrumentation of the target program imposes overhead to the program performance - Only checks the primitive data types rather than objects

2.3.2.7 Monitoring Oriented Programming

Chen et al [Che03][Che04] propose Monitoring Oriented Programming (MOP) framework that supports development and analysis of software through monitoring the formal specification of the software against its runtime behaviour. MOP framework allows the users to specify properties to be monitored in their favourite requirements specification formalism and the recovery actions to be taken if the properties are violated or validated at runtime. This specification is automatically transformed into the monitoring code in a target language and integrated at appropriate places (specified by the user along with the properties and the recovery actions) in the target programme. Any violation and/or validation of a property being monitored at the runtime of the target programme triggers the appropriate action defined by the user. An implementation of the MOP paradigm, called Java-MOP, is presented in [Che05a] [Che05b], which facilitates development and analysis of programmes written in Java. Java-MOP provides a meta-specification language for the user to define specify the monitoring specification. The Java-MOP monitoring specification is divided into three sections. (i) The *heading* section contains the definition of the monitorable events and several configuration attributes of the monitor, including the scope of the monitoring, the exact point of the execution in which properties are checked. (ii) The *body* section contains the properties to be monitored specified in a domain specific requirements specification language. The languages supported by the current implementation of Java-MOP, are JML [Lea00], Jass [Bar01], ERE and LTL [Man95]. (iii) The *handlers* section contains the recovery actions to be taken in case of violation and/or validation of a property. A component, called *logic plug-ins*, transforms the Java-MOP specification into Java monitoring code, which is integrated into the target Java programme by another component, known as *Java annotation processor*.

In addition the target Java programme is also instrumented by using ApectJ [Asp03] to generate runtime events.

Summary

Technique	Java-MOP [Che05a] [Che05b]
Requirement Specification Language Used	JML [Lea00], Jass [Bar01], ERE and LTL [Man95]
Monitorable Event Identification Process	monitrable events are defined by user.
Monitorable Event Specification Language Used	Java-MOP specification language.
Runtime Event Generation Mechanism	Instrumentation
Adaptability	Supported. recovery actions are defined by the user.
Comments	+ Supports new logic addition through logic plug-ins + Allows the user to define recovery actions in case of property violations - Instrumentation of the target program imposes overhead to the program performance - User has to learn Java-MOP specification language in addition to the formal specification language used to define the properties.

2.3.2.8 Requirements Monitoring of Web Service based Systems

Robinson extends the requirements monitoring techniques described in [Fea95, Fea97, Fea98] to deal distributed concurrent transactions [Rob03a]. Robinson uses Goal-driven requirement acquisition process [Dar93] to express high level requirements. For each requirement, potential obstacles are identified by applying KAOS obstacle generation patterns [Lam00]. For each obstacle identified a monitor specification is derived. If the obstacle is directly observable the specification simply assigns the obstacle to an agent for monitoring. To adapt thus approach in distributed environment an architecture is described in [Rob03b]. An *event adaptor* translates web service requests and replies into monitored web service events. A *broadcaster* forwards the monitored events to *listeners*. *Requirements monitor*, which is a specific type of listener, interprets the event stream in terms of requirements satisfaction.

Baresi et al [Bar04a] present a framework for run time monitoring of service compositions expressed as BPEL process. The proposed framework deals with three types of undesirable behaviours, namely *timeouts*, *runtime external errors* and *violations of functional contracts*. A timeout occurs if an activity does not finish in a specific time period. Runtime external

error refers to a failure of an external web service to perform its task. Violation of functional contracts happens if a web service fails to maintain a predefined functional contract, e.g. pre or post condition. A BPEL process is annotated by inserting instructive assertions as comments to instrument the BPEL process. A translator transforms this annotated BPEL process into the monitored BPEL process. To handle timeouts and external error, Baresi et al use BPEL <*eventHandlers*> and <*faultHandlers*> activities respectively. In BPEL <*eventHandlers*> is used to take some action on occurrence of some specific event and <*faultHandlers*> is used to handle internal or external faults. In the proposed framework, for example in case of timeouts, the designer has to insert a comment in the original BPEL process stating the duration of the time-frame and the kind of exception handling that the system should use if the time-out expires. The translator then converts this comment into a BPEL <*eventHandler*> activity with an alarm set to the specific time value in the monitored BPEL process. The proposed framework is further extended in [Bar05a] and [Bar05b, Bar05c], where the monitoring rules are expressed separately rather than annotating BPEL process and the extended framework supports monitoring of QoS properties in addition to the properties described above. In [Bar05a] the constraints to be monitored are expressed as WS-Policy and WS-PolicyAttachment is used to attach the policy to a particular context of the BPEL process. In [Bar05b] the monitoring rules are defined in a *monitoring definition file* that describes the monitoring rules, as well as other monitoring configuration parameters such as priority of a monitoring rule, exact location in the BPEL process where the rule should be associated. Monitoring constraints are expressed in WS-CoL (Web Service Constraint Language), which is a special purpose assertion specification language based on JML [Lea00]. Given the constraints to be monitored in *monitoring definition file* or Ws-Policy and WS-PolicyAttachment, a process weaver instruments the BPEL process. This instrumentation replaces the context of the BPEL process, which a monitoring constraint is applied to, by an invocation to the monitoring component, called *Monitoring Manager*. The *Monitoring Manager* is composed of four components. The *Configuration Manager* holds all the monitoring rules and configuration parameters, the *External Monitor Manager* communicates with the plugins of actual data analyzer, e.g. CLiX[Cli03] or XLINKIT [Xli02], the *Invoker* mainly sends and receives data to and from other components and web services, the *Rule Manager* acts as a co-ordinator and organises the components of monitoring manager. At runtime the instrumented BPEL process invokes the monitoring manager instead of invoking the real web service by passing the data that to be analyzed and the information required to invoke the actual web service. The monitoring manager checks the data with the help of external data analyzers, e.g. CLiX[Cli03] or XLINKIT[Xli02], before invoking the actual web service and also after receiving the response from the invoked web service. If this checking results in a rule violation, a standard exception is raised to the instrumented process. The

positive side of this approach is that, the mechanisms used to handle timeout and external errors can be used to introduce self healing aspect to a Web Service composition process. The mechanism used to monitor the violation of functional contracts would be useful identify malfunctioning Web Services in a composition. However the annotation of BPEL process has negative impact on the process execution as some checks are performed by the service based system itself. Also in case of monitoring functional contracts, single web services are monitored, the overall requirements of the composition process have not been considered.

Lazovik et al [Laz04, Laz06a, Laz06b] propose a framework to plan, execute and monitor business processes. In the proposed framework business processes are modelled as choreography of web services and planning, execution and monitoring of such business process are governed by assertions, which are business rules (user request) applied to the business process. These assertions are classified along two dimensions. The first dimension is known as operation assertions, which is based on the operational context of the assertions. Operation assertions are used to express conditions that must be true in one state before moving to the next state, or to express conditions that must be satisfied throughout all the execution states, or to express properties on the evolution of process variables during process execution. The second dimension of assertions is known as actor assertions, which is based on the ownership of the assertion. Actor assertions are used to express user request applied to the whole business process, or to express assertions applied to all the providers playing a certain role in the process execution or to express assertions applied to a specific provider. The assertions are expressed in XML Service Request Language (XSRL), which is a language to express requests and constraints over requests for web services. Business processes are specified in a choreography language like WS-CDL. The proposed framework is based on interleaving planning and execution of the business process according to the assertions. To realise this, the framework incorporates a *monitor*, a *planner* and an *executor*. The monitor receives the assertions and the business process and contacts the planner for a plan that satisfies the assertions. The planner synthesises a plan and returns it to the monitor. The plan is then executed by the executor step by step. At each execution step the executor also looks for new information about the service implementations. The executor informs the monitor if new information is available for a service. The monitor then contacts the planner for a replan. This iterative process continues until the user request is satisfied according to the given assertions or the planner fails to find a plan that satisfies the user request.

Summary

Technique	Monitoring Web Service requirements [Rob03a, Rob03b]
Requirement Specification Language Used	KAOS assertion
Monitorable Event Identification Process	Formal refinement pattern for goal directed requirements elaboration
Monitorable Event Specification Language Used	-
Runtime Event Generation Mechanism	SOAP messages transmitted between Web services are intercepted and extracted to derive events.
Adaptability	-
Comments	- Web service requests and replies are considered only as runtime events. Internal state of the composition process or the vents internal to the composition process have not been considered.
Technique	Monitoring Web Service Composition [Bar04a] [Bar05a] [Bar05b]
Requirement Specification Language Used	In [Bar04a] Properties to be monitored are expressed as comments in BPEL documents using some predefined constructs. In [Bar05a] WS-Policy framework is used to express monitoring policy and WS-CoL is used to express monitoring rules inside WS-Policy. In [Bar05b] monitoring rules are expressed in WS-CoL.
Monitorable Event Identification Process	-
Monitorable Event Specification Language Used	In [Bar05a] and [Bar05b] WS-CoL is used define event (runtime data) format.
Runtime Event Generation Mechanism	SOAP messages exchanged between web services are used as events.
Adaptability	In [Bar04a] Corrective actions are specified in BPEL process using standard BPEL activities.
Comments	+ Self healing aspect has been introduced to a Web Service composition process. [Bar04a] + Use of standards like WS-Policy to specify monitoring directives. [Bar05a] - Instrumentation of BPEL process imposes overhead to the process performance. [Bar04a] [Bar05a][Bar05b]

2.3.2.9 Monitoring of Service Level Agreements (SLA)

Farrell et al develop an ontology to capture aspects of service level agreements (SLAs) agreed between service provider and consumer [And04a, And04b, And05]. They develop a reasoner to track the states of these contracts against runtime events. Their work is concerned with utility computing (UC) i.e. monitoring of computation resources namely compute power, storage, network bandwidth. The proposed ontology entails three types of contract norms. These are (i) contract management norms that define the effects of contract events on contract

state, (ii) obligation norms that defines the actions a party has to perform in case of violation/fulfilment of the norm, (iii) privilege norms that define the non-contractual actions that party is permitted to perform. An XML based contract language called CTXML, has been devised to formalise the ontology. They used event calculus as computational model and mapped CTXML constructs to XML representation of event calculus (ecXML). An implementation of the computational model is presented, that enables query, written in CTXML or ecXML, of runtime contract states. The implementation also provides a GUI that displays the deployment life cycle of UC SLAs.

Ludwig et al propose an architecture to support the implementation of WS-Agreement [And04] standard, i.e. the architecture aims to establish agreement between a service provider and service consumer and monitor the establish agreement [Lud04]. WS-Agreement driven service management requires three types of functionalities (i) a set of core function to deal the basic WS-Agreement protocol, (ii) a set of domain independent functions that are common to all environments and (iii) a set of domain specific functions in managing services. The proposed architecture covers the first two types of functionalities. For example for the agreement provider it provides components to create agreement template, to decide if an agreement can be accepted or not, to announce new agreement. For the agreement initiator side the architecture provides components to get agreement template instances from the agreement provider and create agreement instances. The work also presents a Java implementation of the architecture.

Summary

Technique	Performance Monitoring of Service-Level Agreements for Utility Computing Using the Event Calculus [And04a] [And04b][And05]
Requirement Specification Language Used	CTXML or ecXML
Monitorable Event Identification Process	
Monitorable Event Specification Language Used	CTXML
Runtime Event Generation Mechanism	External components post contract events via query interpreter.
Adaptability	Remedial actions in case of a contract violation are specified in contractual statements
Comments	+ the implementation can support any contract ontology as long as the ontology can be mapped to ecXML/CTXML - Only non functional requirements are monitored

2.3.2.10 Consistency Checking of Web Service Composition Process

Piccinelli et al [Pic02] describe a lightweight approach to check the consistency of web services composition against some specific business service policies. The interactions involved in a business service usually require the combined use of a number of different web services. A coherent view of the various web services in the composition is essential for the realisation of the overall business service. The suggested framework uses XLINKIT [Nen02] to check consistency of a dynamic set of web services. XLINKIT is a consistency management framework that can be used to check consistency of distributed heterogeneous documents, specially XML-encoded documents. It provides first order logic based language to express constraints between documents and an engine that checks the documents against the constraints. So business service policies can be expressed using XLINKIT and then it can be applied to check consistency among the corresponding WSDL documents describing the Web-Services related to the business service.

Nakajima focuses on the needs to verify web service flows (compositions) prior to their execution, since faulty flow descriptions may be costly in terms of network traffic or useless execution of web services [Nak02a]. According to the author this verification problem is very similar to verification of work flows or business flows. His suggestion is to translate flows into a set of communicating concurrent process and that can be checked by means of software model checking techniques. He also argues that web service flow language should have constructs for exception handling in the presence of accidental failures or malicious attacks. He applies classical software model checking techniques to check reliability of web services flow descriptions by identifying faulty descriptions [Nak02b]. The work uses WSFL as the Web Service flow description language and SPIN [Hol97] as the analyser. WSFL descriptions are translated into Promela and fed to SPIN for verification.

Summary

Technique	Consistency Checking of Web Service Composition [Pic02]
Requirement Specification Language Used	xlinkit rule language is used to describe business service policies.
Monitorable Event Identification Process	-
Monitorable Event Specification	-

Language Used	
Runtime Event Generation Mechanism	-
Adaptability	-
Comments	+ Consistency between component Web Services against some business service policies can be checked before web service composition is made. - Runtime checking of the composition process has not been performed.
Technique	Consistency Checking of Web Service Composition [Nak01, Nak02]
Requirement Specification Language Used	Promela
Monitorable Event Identification Process	-
Monitorable Event Specification Language Used	-
Runtime Event Generation Mechanism	-
Adaptability	-
Comments	+ Faults in web service composition process are identified before it is put in operation. - Runtime monitoring of the composition process has not been performed.

2.3.2.11 Management of Web Service Composition Process

Tosic et al [Tos01] illustrate the need of Web-Services with multiple classes of service and dynamic adaptation of their compositions. Each class of service of one Web-Service offers the same functionality but differ in constraints like authorisation, rights, quality of service and cost. This concept of service offerings (Web-Service with multiple classes of service) is somewhat similar to the well-known concepts of differentiated services in telecommunication and is motivated by the limited underlying resources that the Web-Services use. Since underlying resources are not unlimited, it is suitable to provide different quality of service to different class of Web service's consumers. The paper proposes an extension of WSDL, known as WSOL (Web Service Offering Language), to specify Web-Services with multiple service offerings. WSOL enables specification of functional constraints, non-functional constraints, authorisation policies, cost and other relevant information and constraints. The paper also proposes a management infrastructure, called DAMSC (Dynamically Adaptable and Manageable Service Components), that supports switching between service offerings, deactivation/reactivation of existing service offerings and creation of new appropriate service offerings without breaking the existing relationship between a web service and its consumer. An extended version of WSOL is discussed in [Tos02]. This extended version introduces

some new constructs to define pre-, post- and future- conditions as Boolean expression, service price and penalty (in case of service failure) information etc.

2.3.2.12 UDDI based Management of Web Services

Ali et al point out some limitation of UDDI implementation and propose an extension to UDDI, calls UDDIe, to address those limitations [Ali03]. In addition to the three components in current UDDI, namely white pages, yellow pages and green pages the authors introduce blue pages to record user defined properties associated with a service. Current UDDI does not provide automatic update of the registry as services or service providers change. This may lead to a point where public UDDI registries may contain a lot of listing for services that are no longer active. To resolve this, UDDIe supports finite and infinite leases. In case of finite lease, service provider must define the exact point for which the service should be made available for discovery in the registry. In the current UDDI search for a service can be made only on limited attributes, such as service name, key reference etc. But the blue pages of UDDIe stores user defined service properties such as quality of service (QoS) and the list of methods available within the service that can be called by other services. UDDIe also supports search for services on various properties associated with services, e.g. user defined property, leasing period etc.

Zhou et al implemented an extension of UDDI, UX that facilitates QoS aware discovery of web services [Zho03]. UX works on top of standard UDDI registry and conforms to the UDDI inquiry interface. In the proposed system a local database is maintained to store quality information of individual web service. This quality information is used to predict the future performance of a service. In a typical scenario a service requester defines its preference on the service's QoS metrics (e.g. response time, reliability) using a web based interface and sends a UDDI inquiry to the UX server. The UX server finds the desired service and sends the result back to the requester. The requester uses the service, measures the performance of the service and shares QoS report it makes by sending the report to the UX server. The UX server uses the QoS report to update the quality information of the service in the local database. In the proposed system, the network model is abstracted into domains (e.g. enterprises, universities) and a local UDDI registry works in each domain for web service's discovery. The UX server can perform federated discovery between different co-operating domains if the requested service can't be found in the local domain. The advantage of this approach is that it enables the requester to find a service with desired performance metric without performing any test on the service. While the main disadvantage is that requester has to contribute to the

approach by measuring the performance of the service being used, that imposes overhead to the requester.

2.3.3 Types of Requirements Violations Addressed in the Literature

The approaches reviewed in Section 2.3.2 can detect five different types of violations of requirements at runtime. These types are:

(i) Violation of requirements defined as classical inconsistencies, such violations are treated in [Fea95, Fea98, Coh97, Din02]. Given an assertion (requirement) to be monitored, the assertion is negated and the negated assertion is monitored at runtime. For example, in case of a meeting scheduler system, one requirement is that "the system should know the constraints of all the various participants invited to the meeting within d days". This requirement can be formalised in KAOS as follows,

$$\forall \text{ m: Meeting, p:Participants ConstraintsRequested(p,m)} \Rightarrow \Diamond_{<Xd} \text{ ConstraintsReceived(p,m)}$$

To monitor violations of this requirement, the negation of the assertion that expresses the requirement is used at runtime. This negated assertion is as follows,

$$\exists \text{ m : Meeting, p: Participant } \Diamond \text{ [ConstraintsRequested(p,m)} \wedge \Diamond_{<Xd} \neg \text{ ConstraintsReceived(p,m)]}$$

At runtime if a sequence of events that entails the negation of the assertion is detected, the violation is considered to have taken place.

(ii) Model checking techniques have been exploited in [Fic02a, Fic02b, Fic02c, Pet97, Nak02a, Nak02b]. The requirements specification is transformed into a finite state automaton. The states in the automaton represent acceptable and unacceptable behaviour of the system to be monitored. At runtime the states of the automata are monitored. If the monitor reaches an acceptable state, the requirement is satisfied. If the monitor reaches an unacceptable state or stays too long in an acceptable state, the requirement has failed.

(iii) Violations of *suspect conditions* are treated in [Rob02, Rob03a, Rob03b]. Suspect conditions are weakened conditions of a requirement and are monitored to reduce the cost of requirements monitoring. If *R* is a requirement to be satisfied by the system

and $\neg R \Rightarrow C$, then by observing C at runtime a notification that R is going to fail can be generated. The weakened condition C can be identified by applying refinement patterns for goal-directed requirements elaboration [Dar96].

(iv) QoS properties, such as *timeouts* and *runtime external errors* are monitored in [Bar04a]. A timeout occurs if an activity, within a composite process, does not finish in a specific time period. A runtime external error is defined with respect to a web service composition process, which refers to a failure of an external web service to perform its task.

(v) Satisfiability of non-functional requirements of a system is considered in [Fin01, Efs02, And04a, And04b]. These approaches focus on the monitoring of computation resources such as availability of network bandwidth, consumption of computer power etc.

Although the objective of requirements monitoring is to identify if a requirement fails at the runtime of the system, this fail/no-fail information can be utilised in various useful ways, such as to alert designers, maintainers or users about the redesign of the system or to adjust the system at runtime in order to head off the failure. In the case of (i) and (ii) a deviation is detected only after a requirement has failed. Thus the detection may not be useful in undertaking a corrective action that could fulfil the requirement in the same execution of the system. A corrective action may only be possible to undertake in the same execution of the system if the system supports transaction roll back. However, this information can be used to modify the system to make sure the failed requirement will be satisfied in subsequent executions. On the other hand, in the cases of (iii), (iv) and (v) a warning message signifying the failure of a requirement is generated and this information can be used to take necessary recovery action at runtime and make sure that the requirement that was about to fail will be satisfied.

2.4 Motivation

Web Services technology and research in this field have been developing rapidly since the emergence of this field back in the late nineties. Most of the research in this area is only devoted to the monitoring of QoS properties of Web Services [And04a, And04b, Lud04], or to the composition of Web Services and management of such composition [Nak01, Nak02, Pic02, Ali03, Zho03, Tos01, Laz04, Laz06a, Laz06b]. Functional requirements monitoring of

web services, however, has attracted little attention. [Lud04] focuses on functional requirements of web service composition, but it aims only at the functional requirements of each single web service rather than the overall requirements of the whole composition. Robinson [Rob03a, Rob03b] applies existing software requirements monitoring techniques to monitor web service requirements.

Existing approaches for requirements monitoring have the following limitations:

- Monitorable events are identified by analysing the requirement specification and are subsequently expressed in some event definition language [Fea95, Fea98, Fic02a, Fic02b, Fic02c, Din02]. This manual process of identifying and defining monitorable events is error prone to maintain traceability between the source code and the monitorable event specification.
- Most requirements in a software system are interdependent. Thus monitoring a single requirement in isolation may lead to incorrect result. In some cases monitoring interdependent requirements is necessary. For example, suppose that we have two requirements,

  $$P \wedge Q \Rightarrow R,\ S \Rightarrow P$$

  Suppose also that we want to monitor the first formula, and that the event P is internal to the system and not available to the monitor. Here an occurrence of event P can only be assumed by detecting the event S and then applying deductive reasoning to establish P. So the identification of dependencies between requirements is an essential step in requirements monitoring, as the internal state of systems may not be necessarily known. In such cases it might be necessary to deduce information about this state and may arise in web services where direct access to internal service events may not be possible and the deduced information to detect violations as shown in this example. Cases like these have not been addressed in the literature.

In addition to the above weaknesses, existing requirements monitoring techniques fail to deal adequately with some significant complications, which arise in service-based systems, as they focus on systems with no autonomous components. When, however, such autonomous components exist, as in service-based systems, they can create substantial complexities. These complexities can be summarised as follows,

- In service-based systems, the failure of specific services to function as expected may lead other system components (i.e., services and the system co-ordinating component) to make incorrect assumptions about the state of the system (e.g. the absence of a message confirming the update of some data in one of the system's services does not necessarily mean that these data have not been updated). Consequently, components may take actions, which may be compliant with the requirements but would not have been taken if the correct state of the system was known to them.

Consider, for instance, a car rental system (CRS) which acts as a broker offering its customers the ability to rent cars provided by different car rental companies directly from car parks at different locations. Suppose also that CRS is implemented as a service based system that consists of a service composition process that interacts with:

- Car information services (IS) which are provided by different car rental companies, and maintain registries of cars, check car availability and allocate cars to customers as requested by CRS.
- *Sensoring services* (SS) which are provided by different car parks to sense cars as they are driven in or out of car parks and inform CRS accordingly.
- *User interaction services* (UI) that provide CRS with a front-end that handles interactions with the end-users.

In a typical operational scenario, CRS receives car rental requests from UI services and checks for the availability of cars by contacting IS services. If an available car can be found at the requested location, CRS books the car rental through an IS service, and takes payment. When cars move in and out of car parks, SS services inform CRS, which subsequently invokes operations in IS services to update the availability status of the moved car. In this scenario, CRS may wrongly accept a car rental request and allocate a specific car to it if, due to malfunctioning of an SS service, the departure of the relevant car from a car park has not been reported and, as a consequence, the car is considered to be available by IS.

The five types of inconsistencies addressed in the literature and discussed in Section 2.3.3 are not enough to handle scenarios described above. Covering for possibilities like this requires the introduction of types of requirements deviation beyond classical inconsistency and the development of appropriate reasoning mechanisms for detecting them.

- Instrumentation has been used to generate run time events in [Rob02, Kan00, Kim01, Din02]. In service-based systems the generation of events through code instrumentation may not be possible as the provider of the system might not have ownership of the individual services which constitute it and individual services may change dynamically. In CRS, for example, typically the SS and IS services won't be owned by the owner of CRS. In addition, new instances of these services may be deployed when new car rental companies and car parks make their offerings available to the system, and existing instances may be withdrawn when companies and car parks stop their collaboration with CRS. Also, although reflection could provide a general-purpose mechanism for obtaining information about the state of individual services at run-time, it cannot be guaranteed that all individual services will be implemented in languages with reflective capabilities. Thus, monitoring may have to be based on events and state information that can be obtained from the system co-ordinating component which can be reasonably assumed to be in the ownership of the system provider. This restriction makes most of the existing requirements monitoring techniques not applicable in the case of service-based systems. In such systems, requirements for individual services may still be specified and monitored but only if this is possible through events, which are known to the co-ordinating component of the system, either because they can be captured directly at runtime or because they can be deduced from runtime events.

- Requirements often specify temporal constraints over the behaviour of a system (e.g., a system might be required to produce a response following the occurrence of a specific event within a given time period). In service-based systems, the specification and checking of these constraints must take into account the time required for the communication between the interacting services. This time, however, is not negligible as it is typically assumed in requirement specifications of centralised systems, and may vary depending on the physical distribution of the services on different processors and/or network communication delays.

2.5 Our Approach

In this section we present the basic aspects of our approach to build a monitoring framework that could address the problems identified in Section 2.4 and that can be applied to monitor requirements of Web Service based systems. Our approach assumes that a service-based system is built as a collection of web-services, which are co-ordinated by a composition process specified in BPEL [Bpe03]. It also assumes that, at run-time, a process execution

engine executes the BPEL composition process and delivers the functionality of the system. Our framework accepts a monitoring policy as the input to the monitoring process. This monitoring policy is expressed in XML and contains the specification of the formulas to be monitored. We have defined a language to specify formulas and the language has a formal grounding on *event calculus* (EC) [Sha99]. The monitoring policy may include four types of formulas, namely (i) *Behavioural Properties* (ii) *Functional Properties* (iii) *Quality of Service (QoS) Properties* and (iv) *Assumptions*. These formula types are defined below:

(i) *Behavioural properties* – These represent alternative paths of the service composition process of a service based system, and enable the monitoring of their execution at run-time. These properties are automatically extracted from the specification of the composition process in BPEL (see Section 2.5.1). Behavioural properties are checked against the stream of events that indicates the behaviour of a service based system at runtime in order to establish whether they are violated.

(ii) *Functional Properties* – These properties express functional requirements for the individual services of a service based system or groups of such services such as pre-conditions and post-conditions that must be satisfied before and after the execution of operations of individual services. Functional properties are also checked against the stream of events that indicates the behaviour of a service based system at runtime in order to establish whether they are violated.

(iii) *Quality-of-service* properties – These properties express quality requirements for individual services or groups of services such as reliability and performance requirements for individual services or paths of the service composition workflow, acceptable rates of denials of service by individual service operations and acceptable rates of service availability. Like functional and behavioural properties, quality of service are checked against the stream of events that indicates the behaviour of a service based system at runtime in order to establish whether they are violated.

(iv) *Assumptions* – These formulas express conditions that are used to generate additional information about the expected service behaviour and its effect on the state of the system. Assumptions are not checked at runtime against the stream of events that indicates the behaviour of a service based system. Instead, they are used only to deduce information about the state of the individual services which are deployed by this system. In this capacity, assumptions constitute an effective mechanism for getting information about the internal state of the services of a service based system

in cases where the instrumentation of such services is not allowed (due to service ownership restrictions) or desired.

At run-time, the proposed framework obtains event occurrences by catching events which are exchanged between the individual services and the co-ordinating component of the system without requiring the modification of the code that implements these services or the existence of reflective capabilities in them. Thus our approach is *non intrusive* from this aspect, i.e. it performs the monitoring as a computational entity that is external to the system that is being monitored. Monitoring is carried out in parallel with the operation of this system and does not intervene with this operation in any form. The compliancy of the intercepted events is then verified against the properties. The monitoring process checks whether the runtime behaviour of the service based system violates its behavioural properties or the functional and QoS requirements set for it. During this process, assumptions are used to generate additional information about the expected service behaviour and its effect on the state of the system. This effect is represented by special state variables that the users introduce. These external state variables are different from the state variables defined in the composition process of the system and enable the specification of properties for service based system (see Chapter 3). The framework supports detection of different types of violations of requirements. These types include

(i) violations of functional properties and quality-of-service properties by the recorded behaviour of the service based system.

(ii) violations and potential violations of behavioural properties, functional properties and quality-of-service properties by the expected system behaviour (i.e. the behaviour that would have been exhibited by the system if all the functional properties and assumptions assumed regarding the behaviour of individual services had been satisfied)

(iii) cases of unjustified and potentially unjustified system behaviour that may arise due to incorrect information about the state of the system that has led to the execution of incorrect service orchestration paths.

Figure 2.15 shows the architecture of our monitoring framework [Mah04][Spa06]. As shown in this figure this architecture incorporates eight main components. These components are: a *behavioural properties extractor*, an *event receiver*, a *monitor manager*, a *monitor*, a

monitoring console, a *simulator,* an *event database handler* and a *formula database handler.* The role and functionality of these components are overviewed below.

2.5.1 Behavioural Properties Extractor

The behavioural properties extractor extracts the behavioural properties to be monitored from the specification of the composition process of service based systems expressed in BPEL. Behavioural properties are extracted according to extraction patterns designed and are represented in the XML-based language that we have defined to represent event calculus formulas. The extraction mechanism is described thoroughly in Chapter 5. As a by-product of this extraction, the properties extractor identifies events in the BPEL composition process of a service based system which provide the primitive ingredients for specifying functional properties, quality of service properties and assumptions. Functional properties, assumptions and quality of service properties can be specified in terms of these monitorable event types by the system provider using the *property editor* incorporated by the *monitoring console.*

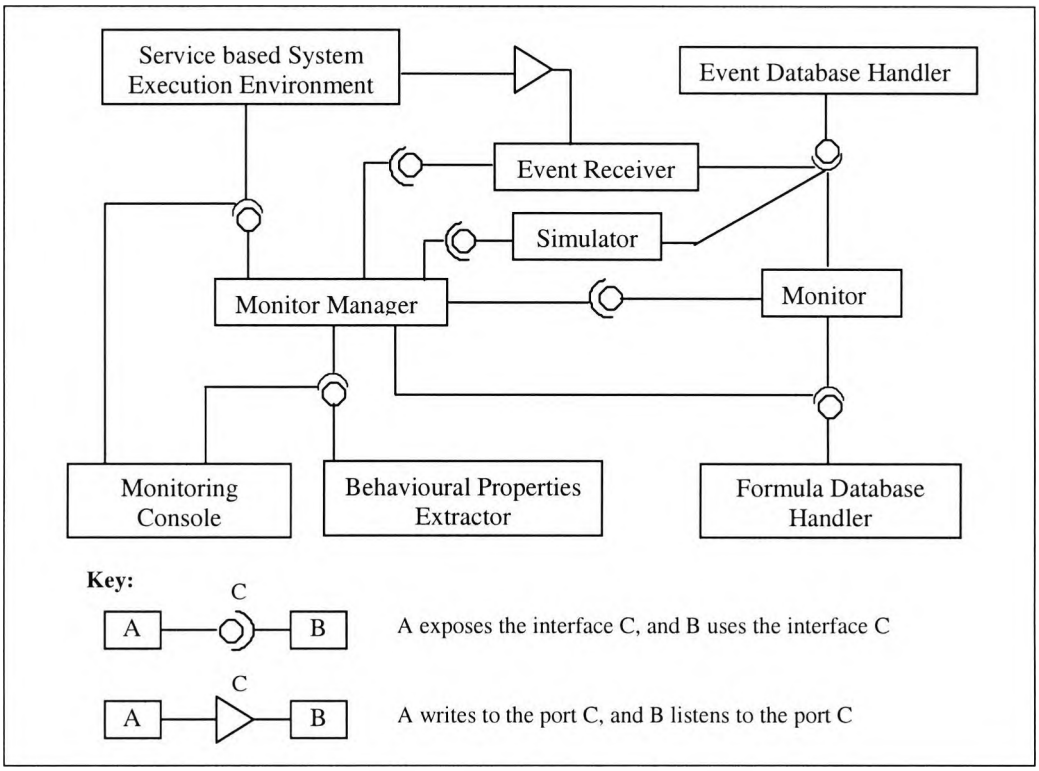


Figure 2.15: Architecture of the monitoring framework

2.5.2 Event Receiver

While executing the composition process of a service based system, the process execution engine generates events, which are sent as string streams to the *event receiver* of our framework. The event receiver identifies the type of the events that its input stream describes, filters out events, which are irrelevant to the monitoring process and records all other events in an *event database*. The events of the process execution engine, which are irrelevant, are determined by the formulas that have been extracted or specified for monitoring by the system provider. The mechanisms used to generate monitoring events are discussed in Chapter 5.

2.5.3 Monitor

The *monitor* processes the events, which are recorded in the *event database* by the event receiver in the order of their occurrence, identifies other expected events that should have occurred but have not been recorded (these are events that can be derived from the functional properties and assumptions that the individual services of service based system are required to satisfy), and checks if the recorded and expected events are compliant with the properties which must be monitored for a system. In cases where a property is not consistent with the recorded and the derived events, the monitor generates a *deviation report* and records it in a database. Information about deviations is recorded in XML. The monitoring algorithms applied by the monitor are described in detail in Chapter 4.

2.5.4 Monitoring Console

The architecture of Figure 2.15 incorporates also a *monitoring console* that gives access to the monitoring service to human users. The monitoring console incorporates a *property editor* that presents system providers with predicates that signify the different types of monitorable events that have been identified in the BPEL process of a service based system. System providers can specify functional properties, quality of service properties and assumptions as logical combinations of these event predicates. The property editor provides a graphical user interface for specifying properties and checks their syntactic correctness. The monitoring console also contains a *deviation viewer* that displays the deviations from the monitored requirements. The user can browse the detected violations of the formulas and view the details of each formula violation using the deviation viewer.

2.5.5 Monitor Manager

The *monitor manager* is the component that has responsibility for the initiation and coordination of the monitoring process and reporting its results. Once it receives a request for starting a monitoring activity as specified by a monitoring policy, it checks whether it is possible to monitor the requirements specified in this policy given the BPEL process of the service based system that is identified in the policy, and the event reporting capabilities indicated by the type of the execution environment of the service based system. If the requested properties can be monitored, it starts an event receiver to capture events from the service based system execution environment and passes to it the events that should be collected. It also sends to the monitor the formulas to be checked.

2.5.6 Simulator

The simulator is a component in our framework that can be used to generate monitoring events by simulating a BPEL process. Hence it allows the system provider to monitor a simulated BPEL process. Simulation can support the estimation of the time required for monitoring specific properties before starting monitoring them against the real system operations and assists system providers in making decisions about the allocation of such properties to different monitors depending on such time estimates. A detail description of the simulator is presented in Chapter 5.

2.5.7 Formula Database Handler

The *formula database handler* maintains the communication between the formula database and the other components of the monitoring framework. It allows the *monitor* to store formula instances in the database. It also allows the *monitor* and the *monitor manager* to retrieve formula instances from the database.

2.5.8 Event Database Handler

The *event database handler* maintains the communication between the event database and the other components of the monitoring framework. It allows the *event receiver* and the *simulator* to store events in the event data base. It also allows the *monitor* to retrieve events from the event database.

Chapter Three

Specification of Monitoring Policies

3.1 Overview

Our monitoring framework has been designed with the objective to support two different monitoring scenarios for service based systems using a non intrusive approach. In the first of the assumed monitoring scenarios (referred to as "Scenario 1" henceforth), a human user (typically the provider of a service based system) can directly request the framework to monitor whether the runtime operation of the system satisfies certain properties and view any deviations from these properties as soon as they are detected. In the second scenario (referred to as "Scenario 2" henceforth), the monitoring can be requested by the environment that executes the composition process of a service based system or a computational entity acting on its behalf for a certain period of time (or until further notice). In this scenario the service based system execution environment or the computational entity that has requested the monitoring has to poll the framework to retrieve any deviations of the properties which are being monitored. In both these scenarios, the input to the monitoring framework is a *monitoring policy* that contains the formulas to be monitored and other monitoring parameters.

In Section 3.2 we introduce the monitoring policies used by our framework and define a schema to specify such monitoring policies. In Section 3.3 we introduce the formal language for specifying the monitorable properties and assumptions in our framework. This language is based on event calculus [Sha99] and is called EC-Assertion. EC-Assertion is an XML based language that is defined by an XML Schema that is founded upon the formal framework of Event Calculus. Thus, it allows the specification of monitorable properties in a form that makes it possible to check the validity of the formulas, process them and exchange the formulas between different tools.

3.2 Policy Specification

As discussed in Section 3.1, the input to our monitoring framework is a monitoring policy. This policy contains the formulas to be monitored and other monitoring parameters. More specifically the policy specifies:

(i) The BPEL process of the service based system to be monitored and the WSDL specifications of the web-services deployed by this process.

(ii) The formulas that should be monitored at runtime. Formulas in the monitoring policy is optional in case of *Scenario 1*, since the system provider can use the monitoring framework to extract the behavioural properties, and also to specify functional properties, quality of service properties and assumptions.

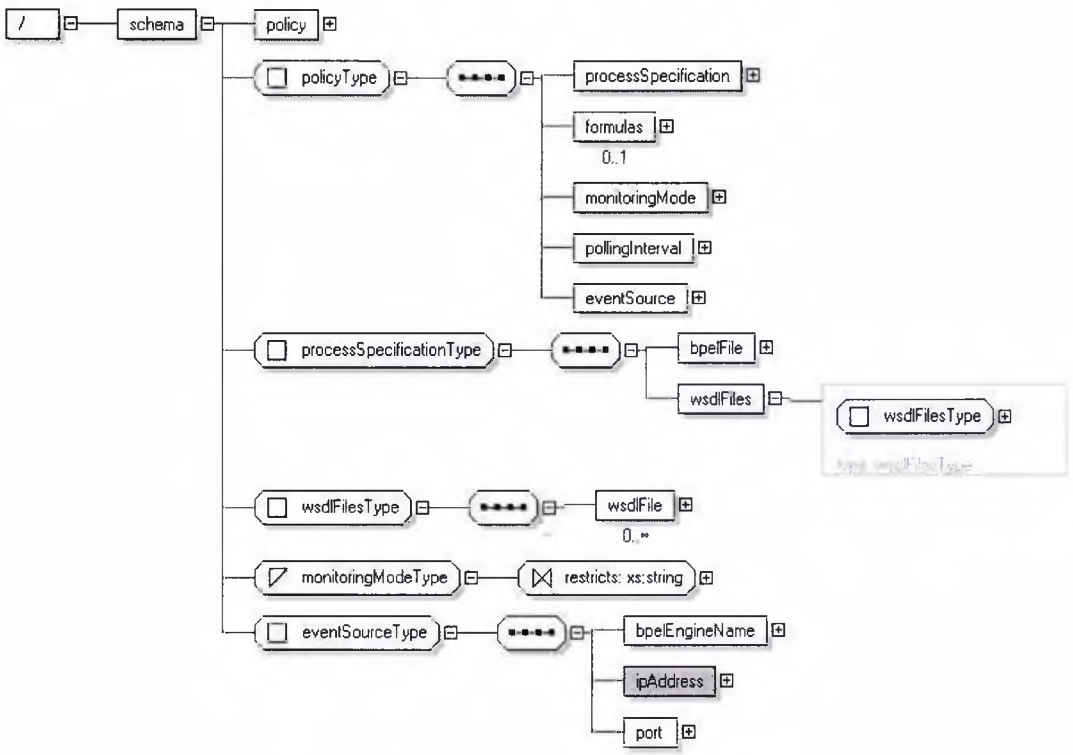


Figure 3.1: Graphical view of the monitoring policy schema

(iii) The mode of monitoring, i.e. whether the monitoring should be with respect to recorded events only, or with respect to mixed (both recorded and derived) events. See Chapter 4 for clarification.

(iv) The source of the events that will provide the information on which monitoring will be based. The source of the runtime events is specified by an IP address and a port

number where the service based system's execution environment will deliver the stream of events at runtime to allow their collection by the monitoring framework.

(v) The mode of reporting any deviations of the monitored properties. The reporting mode of a policy specifies the time between the generations of consecutive reports of deviations.

To enable the specification of monitoring policies we have defined an XML schema [Xml04a], a graphical view of which is shown in Figure 3.1. The textual description of the elements of the policy schema is shown in Table 3.1. The complete schema is shown in Appendix B.

Table 3.1. Textual description of the elements of the policy schema

Element	Description
<xs:element name="policy" type="policyType"/>	This is the element that is used to define a monitoring policy in the XML document. It has type *policyType*.
<xs:complexType name="policyType"> <xs:sequence> <xs:element name="processSpecification" type="processSpecificationType"/> <xs:element name="formulas" type="fns:formulasType" minOccurs="0"/> <xs:element name="monitoringMode" type="monitoringModeType"/> <xs:element name="pollingInterval" type="xs:long"/> <xs:element name="eventSource" type="eventSourceType"/> </xs:sequence> </xs:complexType>	This element defines the structure of a policy and has following child elements: (i) a child element of type *processSpecificationType* called *processSpecification* which is used to describe the BPEL process to be monitored, e.g. BPEL file name, WSDL file names. (ii) an optional child element of type *formulasType* called *formulas*, which is used to describe the formulas to be monitored. This type is not defined within the policy schema. It is defined as part of the schema for expressing monitorable properties (see Section 3.3.5 for the definition of this type), (iii) a child element of type *monitoringModeType* named as *monitoringMode* which is used to describe the monitoring mode. (iv) a child element of type *long* called *pollingInterval,* which is used to specify the interval between the generations of consecutive reports. (v) a child of type *eventSourceType,* called *eventSource,* which is used to describe the event source.
<xs:complexType name="processSpecificationType"> <xs:sequence> <xs:element name="bpelFile" type="xs:string"/> <xs:element name="wsdlFiles" type="wsdlFilesType"/> </xs:sequence> </xs:complexType>	This element defines the BPEL process to be monitored. It has the following child elements: (i) a child element of type *string* called *bpelFile*, which is used to specify the reference to the BPEL file for the process to be monitored. (ii) a child element of type *wsdlFilesType* called *wsdlFiles,* which is used to specify a list of WSDL files involved in the BPEL process to be monitored.
<xs:complexType name="wsdlFilesType"> <xs:sequence> <xs:element name="wsdlFile" type="xs:string" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType>	This element specifies a list of references to WSDL files. It has zero or more child elements of type *string* called *wsdlFile,* each of these refers to a WSDL file.
<xs:simpleType	This element defines the monitoring mode of a

name="monitoringModeType"> <xs:restriction base="xs:string"> <xs:pattern value="recorded|mixed"/> </xs:restriction> </xs:simpleType>	policy. This mode can be either *recorded* or mixed and signifies the type of the events which should be used for monitoring.
<xs:complexType name="eventSourceType"> <xs:sequence> <xs:element name="bpelEngineName" type="xs:string"/> <xs:element name="ipAddress" type="xs:string"/> <xs:element name="port" type="xs:int"/> </xs:sequence> </xs:complexType>	This element is used to describe the event source, i.e. the component which will provide the monitoring events and has the following child elements: (i) a child element of type *string* called *bpelEngineName* which is used to specify the type of BPEL engine. (ii) a child element of type *string* called *ipAddress,* which is used to specify the IP address of a host. (iii) a child element of type *int* called *port* which is used to specify a port number in the host machine where the runtime events are sent by the BPEL engine.

3.3 Property Specification

As discussed in Chapter 2, in our monitoring framework behavioural properties, functional properties, QoS properties and assumptions are expressed in a language based on event calculus [Sha99]. In this section we define this language which is named as EC-Assertion. We introduce event calculus, explain the motivation behind the use of event calculus to represent properties in our framework, present the types of events and fluents used in our framework and describe the formula structure. We also describe a schema that allows the representation of event calculus formulas in XML.

3.3.1 The Basics of Event Calculus

Event calculus (referred to as "EC" in the rest of this thesis) is a temporal logic language based on first-order predicate calculus that can be used to represent and reason about the behaviour of dynamic systems. In EC, this behaviour is specified in terms of *events* and *fluents*.

An event is something that occurs at a specific instance of time and may change the state of a system. A fluent is any property of the system whose value is subject to change over time and fluent is a signifier of a system state. EC allows the specification of system events and the time when they occur. It also allows the specification of initialisations and modifications of system states in response to these events at specific times. In our framework, the specification of behavioural properties, functional properties, QoS properties and assumptions uses the following predicates:

- **Happens**(e,t,$\Re$(t1,t2)) – This predicate is a special case of the *Happens* predicate of standard EC (see [Sha99]) signifying the occurrence of an event e of instantaneous duration at some time t that is within the time range $\Re$(t1, t2). This time range must be specified for each time variable appearing in a *Happens* literal. Thus, the predicate **Happens**(e,t,$\Re$(t1,t2)) in our framework is equivalent to the formula:
  Happens'(e,t) $\wedge$ (t1 $\leq$ t) $\wedge$ (t $\leq$ t2)
  where **Happens'**(e,t) is the predicate that signifies an event occurrence in standard event calculus.

- **Initially**(f) – This is a standard predicate of EC signifying that a fluent f holds at time 0 (i.e. at the beginning of a theory).
- **Initiates**(e,f,t) – This is a standard predicate of EC signifying that a fluent f starts to hold after the event e at time t.

- **Terminates**(e,f,t) – This is a standard predicate of EC signifying that a fluent f ceases to hold after the event e occurs at time t.

- **HoldsAt**(f,t) – This predicate signifies that the fluent f holds at time t.

- **Clipped**(t1,f,t2) – This is a predicate signifying that a fluent f ceases to hold at some time instance between t1 and t2. It differs from the standard EC *Clipped(t1,f,t2)* in that in standard definition the boundaries t1 and t2 are not inclusive, but in our framework the boundaries t1 and t2 are inclusive. This is because we are using the special case of the *Happens* predicate introduced above and the *Clipped* predicate is defined using this special *Happens* predicate (see axiom EC1 and EC9 below).

- **Declipped**(t1,f,t2) – This is a predicate signifying that a fluent f starts to hold at some time instance between t1 and t2. It differs from the standard EC *Declipped(t1,f,t2)* in that in standard definition the boundaries t1 and t2 are not inclusive, but in our framework the boundaries t1 and t2 are inclusive. This is because we are using the special case of the *Happens* predicate introduced above and the *Declipped* predicate is defined using this special *Happens* predicate (see axiom EC2 and EC9 below).

A specification in our framework must also be compliant with the following domain independent axioms[1]:

(EC1) Clipped(t1,f,t2) ⇐ (∃e,t)Happens(e,t,ℜ(t1,t2)) ∧
Terminates(e,f,t)

The axiom EC1 states that a fluent f is clipped (i.e. ceases to hold) within the time range from t1 to t2, if an event *e* occurs at some time point t within this range and *e* terminates f.

(EC2) Declipped(t1,f,t2) ⇐ (∃e,t) Happens(e,t,ℜ(t1,t2)) ∧
Initiates(e,f,t)

The axiom EC2 states that a fluent f is declipped (i.e. it comes into existence) at some time point within the time range from t1 to t2, if event e occurs at some time point t, between times t1 and t2 and fluent f starts to hold after event e at t.

(EC3) HoldsAt(f,t) ⇐ Initially(f) ∧ ¬Clipped(0,f,t)

The axiom EC3 states that a fluent f holds at time t, if it held at time 0 (i.e. at the beginning of a theory) and has not been terminated between times 0 and t.

(EC4) HoldsAt(f,t2) ⇐ (∃e,t1) Happens(e,t1,ℜ(t1,t2)) ∧
Initiates(e,f,t1) ∧ ¬Clipped(t1,f,t2)

The axiom EC4 states that a fluent f holds at time t2, if an event e has occurred at some time point t1 before t2 which initiated f at t1 and f has not been clipped between t1 and t2.

(EC5) ¬HoldsAt(f,t2) ⇐ (∃e,t1) Happens(e,t1,ℜ(t1,t2)) ∧
Terminates(e,f,t1) ∧ ¬Declipped(t1,f,t2)

The axiom EC5 states that a fluent f does not hold at time t2, if there is an event e that occurred at some time point t1 before t2 which terminated fluent f and this fluent has not been declipped at any time point from t1 to t2.

(EC6) HoldsAt(f,t2) ⇐ HoldsAt(f, t1) ∧ t1 < t2 ∧ ¬Clipped(t1,f,t2)

[1] All variables are assumed to be universally quantified in the axioms, unless specified explicitly.

The axiom EC6 fluent f holds at time t2, if it held at time t1, where time point t1 is before time point t2 and fluent f has not been clipped between times t1 and t2.

(EC7) $\neg$HoldsAt(f,t2) $\Leftarrow$ $\neg$HoldsAt(f, t1) $\wedge$ (t1 < t2) $\wedge$
$\neg$Declipped(t1,f,t2)

The axiom EC7 states that a fluent f does not hold at time t2, if it did not hold at some time point t1 before t2 and f has not been declipped since then.

(EC8) Happens(e,t,$\Re$(t1,t2)) $\Rightarrow$ (t1 $\leq$ t) $\wedge$ (t $\leq$ t2)

The axiom EC8 states that the time range in a Happens predicate is inclusive of its boundaries.

EC1-EC7 are axioms of the standard EC (see [Sha99]). EC8 is an axiom that is introduced in our framework to restrict the validity of the time ranges specified for *Happens* literals.

3.3.2 Why Event Calculus

The selection of event calculus as the basis of the property specification language of our framework has been motivated by the need to express the properties to be monitored in a formal language with well-defined semantics allowing:

(a) Reasoning based on the inference rules of first-order logic. This is because studies showed that first order logic has adequate expressive power to specify properties for wide range of applications without increasing the complexity of the logic [Bel00] [Ijc04].

(b) The specification of temporal constraints, which are essential in specifying and verifying temporal aspects of the execution of computer programs [Lam80] [Lam83], and therefore servcie based systems which are the focus of this thesis.

Furthermore, EC offers several advantages over other formal languages from certain points of view. More specifically,

- During the execution of a system events occur instantaneously, whereas states represent information that holds for a duration of time. The distinction between events and states is

very important for monitoring a system execution as it provides a natural way for describing their behaviour. Our claim is based on the observation that typically even non logic based specifications of software system behaviour are based on automata which have a very similar notion of states and events that cause transitions (changes) between states. EC provides a clear distinction between the events and fluents (states) of a system by introducing a limited set of specific predicates unlike other temporal logic languages, such as Computation Tree Logic (CTL) [Cla86][Var98] [Var01], Linear Time Logic (LTL) [Man95] [Var98] [Var01], Propositional Temporal Logic (PTL) [Gab80] [Bel00] or other variants of temporal logic languages [Tho91] [Dar93][Dar96][Pin94], which allow the introduction of predicates with arbitrary meanings.

- The specification of changes of system states in expressing monitorable properties is also significant, since our monitor uses state information to reason about the state of the execution of the system. In addition to the clear distinction between events and fluents, EC has a set of specific predicates (see Section 3.3.1) that signify the occurrence of events and their effect on the initiation/termination of fluents (states).

- In most of the alternative temporal logic languages, including CTL [Cla86] [Var98] [Var01], LTL [Man95] [Var98] [Var01], PTL [Gab80] [Bel00] and other variants [Dar93] [Dar96] [Pin94], temporal constraints are expressed using temporal operators, which are qualitative in nature and it is not possible to put specific boundaries on time [Bel00]. Unlike these languages, EC has an explicit time structure that allows users to specify complex quantitative temporal relationships, such as temporal distances between events and constraints regarding the duration of events in time units.

- EC offers a linear temporal structure that provides a more natural representation for reasoning [Den95] [Den96], compared to some other temporal logic with branching time like CTL [Cla86] [Var98] [Var01]. Moreover the time structure in EC enables the expression of both future and past properties, which is not permitted in some temporal languages (e.g. PTL [Gab80] [Bel00]).

- EC enables a more detailed representation of a process than other event-fluent centric languages like situation calculus [Lev98] [Sow03]. For example multiple states and events can be combined into a single situation in situation calculus, but EC always defines the influences between individual events and fluents.

- Unlike pure state-transition representations, EC has an explicit time structure that does not depend on any sequence of events under consideration. This feature of EC allows to model a wide range of event driven systems, e.g. systems for which state space is infinite [Mil99][Ale02]

3.3.3 Fluents and Events

Our EC based language uses special types of events and fluents to specify properties of SBS systems. These types of fluents and events are described below.

3.3.3.1 Fluents

A fluent in our framework takes the form

valueOf(fluent_var, value_exp) (I)

The meaning of this fluent is that the variable signified by *fluent_var* has the value *value_exp*. When the fluent (I) appears in an *Initiates* predicate its meaning is that *fluent_var* is assigned the value of *value_exp*. When it appears in a *HoldsAt* predicate it denotes an equality check over the values of *fluent_var* has the value *value_exp*, that is whether the value of *fluent_var* is equal to the value *value_exp*.

In fluent (I),

- *fluent_var* denotes a typed variable or a list of typed variables. The variable denoted by *fluent_var* may be an *internal* or an *external* variable. An internal variable is a variable in the composition process of an SBS system. An *external variable* is a variable introduced by the user to represent the state of the SBS system at run-time. If *fluent_var* has the same name as a variable in the SBS composition process then it denotes this variable, and is treated as an internal variable. In all other cases, *fluent_var* denotes an external variable.

- *value_exp* is a term that represents either a variable in the logic language of EC or a call to an operation that returns an object of some type. The operation called by *value_exp* may be a built-in operation of the monitoring framework or an operation that is provided by an external web-service. If *value_exp* signifies a call to an operation, it can take one of the following two forms:

- *oc:S:O(_Oid,_P1,...,_Pn)* that signifies the invocation of an operation *O* in an external service *S*.
- *oc:self:O(_Oid,_P1,...,_Pn)* that signifies the invocation of the built-in operation *O* of the monitor.

In these forms,

- *_Oid* is a variable whose value identifies the exact instance of *O*'s invocation within a monitoring session, and
- $_P_1$, ..., $_P_n$ are variables that indicate the values of the input parameters of the operation *O* at the time of its invocation.

The internal operations which may be used in the specification of fluents are shown in Table 3.2.

Table 3.2: Built-in operations for properties specification

Operation	Description
add(n1:Real, n2:Real): Real	This operation returns n1+n2
sub(n1:Real, n2:Real): Real	This operation returns n1-n2
mul(n1:Real, n2:Real): Real	This operation returns n1* n2
div(n1:Real, n2:Real): Real	This operation returns n1/n2
append(a[]: list of <T>, e:T): list of <T> where T is Real, Int or String.	This operation appends e to a[].
del(a[]: list of <T>, e:T): list of <T> where T is Real, Int or String.	This operation deletes the first occurrence of e in a[].
delAll(a[]: list of <T>, e:T): list of <T> where T is Real, Int or String.	This operation deletes all occurrences of e in a[].
size(a[]: list of <T>): Int where T is Real, Int or String.	This operation returns the number of elements in a[].
max(a[]: list of <T>):<T> where T is Real, Int or String.	This operation returns the maximum value in a[].
min(a[]: list of <T>):<T> where T is Real, Int or String.	This operation returns the minimum value in a[].
sum(a[]: list of <T>):<T> where T is Real or Int.	This operation returns the sum of the values in a[].
avg(a[]: list of <T>): <T> where T is Real or Int.	This operation returns the average of the values in a[].
median(a[]: list of <T>):<T> where T is Real, Int or String.	This operation returns the arithmetic median of the values in a[].
mode(a[]: list of <T>): <T> where T is Real, Int or String.	This operation returns the most frequent element in a[].
new(type_name: String): ObjectIdentifier	This operation creates a new object instance of type T and returns an atom that is a unique object identifier for this object.

Calls to external and internal operations in fluents allow us to provide complex computations which are necessary for checking certain properties within the reasoning process of the monitoring framework (e.g. to compute the average or standard deviation of a series of response times).

Typing Conditions for Fluents

A fluent expression *valueOf(fluent_var, value_exp)* is valid if and only if the type of *fluent_var* is a subtype of the type of *value_exp*. The type of *fluent_var* and *value_exp* can be either a primitive type (e.g. integer, string) or WSDL message type. We represent a WSDL message type as a tree, since WSDL message is a part of XML document (i.e. WSDL document) and XML can essentially be represented as tree [Xml97] [Abi97] [Min05]. The tree is constructed by considering the BPEL variable that refers to WSDL message as the root, primitive parts of the WSDL message as the leaf nodes and complex parts of the WSDL message as non leaf nodes of the tree. Figure 3.2 shows a tree that represents a BPEL variable. In the tree structure in Figure 3.2 the root is shown as rounded rectangle, non leaf nodes are shown as circles and leaf nodes are shown as rectangles. A leaf is accessed by combining all the part/message names from the root to the leaf, i.e. *areaCode* in the tree in Figure 3.2, is accessed as *contact.phone.areaCode*.

WSDL message definition

```
<definitions xmlns:tsns=http://www.complex.types/ .....>
  .....
  <types>
   <xsd:schema
       targetNamespace="http://www.complex.types/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:complexType name="phoneType">
       <xsd:element name="areaCode" type="xsd:int"/>
       <xsd:element name="number" type="xsd:int"/>
     </xsd:complexType>
   </xsd:schema>
  </types>
  <message name="ContactType">
    <part name="name" type="xsd:string "/>
    <part name="phone" type="tsns:phoneType"/>
  </message>
  .....
</definitions>
```

BPEL variable declaration

```
<process name="X"xmlns:vns=http://www.complex.types/ ..>
.....
  <variables>
    <variable name="contact"
          messageType="vns:ContactType"/>
  </variables>
  .....
</process>
```

Tree representation of the BPEL variable

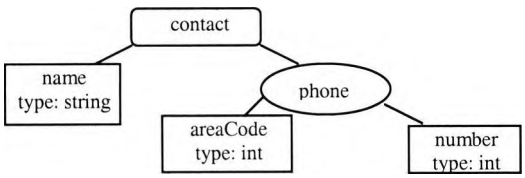


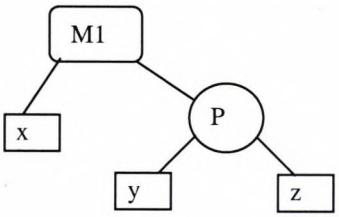
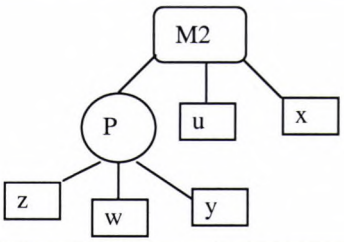
Figure 3.2: Tree representation of BPEL variable

Given the tree representation of BPEL variable, the fluent *valueOf(fluent_var, value_exp)* is valid as long as *fluent_var* is a sub tree isomorphism of the *value_exp*. Sub tree isomorphism is defined as follows,

"Given two rooted trees G and H, there is a sub graph G' of G (whose root is the root of G) such that there is an isomorphism between H and G' that maps the root of H to the root of G' " [Tsu99] [Gib90].

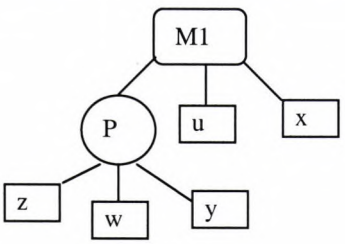
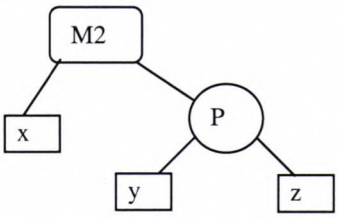
Given the above definition of sub tree isomorphism, we explain the type validity condition of *valueOf(fluent_var, value_exp)* with some examples below,

Case 1:

Fluent_var	Value_exp
	

In the case of the trees for the variables M1 and M2, a fluent of the form *valueOf(M1, M2)* is valid, since *M1.x*, *M1.P.y* and *M1.P.z* can be mapped to *M2.x*, *M2.P.y* and *M2.P.z* respectively.

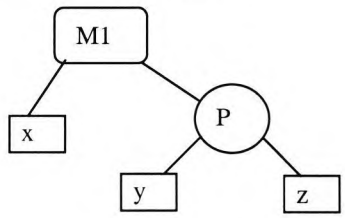
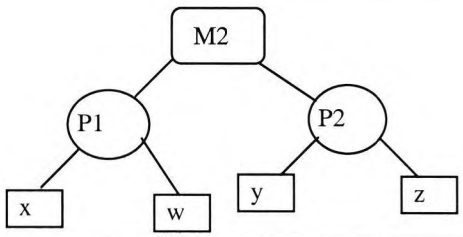
Case 2:

Fluent_var	Value_exp
	

In the case of variables M1 and M2 of case 2, a fluent of the form *valueOf(M1, M2)* is not valid, since there is no leaf in the tree representation of the *value_exp* (M2) that *M1.u*, *M1.P.w* can be mapped to.

Case 3:

In the case of variables M1 and M2 of case 3, a fluent of the form *valueOf(M1, M2)* is not valid, since *M1.x* can not be mapped to *M2.P1.x* as they are not at the same level of the respective tree.

Fluent_var	Value_exp
	

If *fluent_var* denotes an external variable (i.e. a variable that is introduced by the user), the specification of its type is deduced from the type of *value_exp* in a fluent specification. In this case, if the variable appears in different fluents that use different *value_exp* terms, the above type validity condition should be satisfied by the types of all the relevant *value_exp* terms. If *fluent_var* denotes an internal variable, its type is determined by the specification of the relevant variable in the composition process of the SBS system. In this case, the type validity condition must be satisfied by the types of all the expressions *value_exp* that co-exist with *fluent_var* in fluent specifications.

3.3.3.2 Events

Events in our framework represent exchanges of messages between the composition process of an SBS system and the services co-ordinated by it and the assignment of values to internal variables of the composition process of an SBS system. More specifically, events can be of one of the following five types:

(i) *Service operation invocation events* – These are events that signify the invocation of an operation in one of the partner services of an SBS system by the composition process of it. Events of this type are represented by terms of the form

$$in{:}S{:}O\ (\_Oid,\ \_P_1,\ \ldots,\ \_P_n)$$

where

- *in* signifies this is a service operation invocation event;
- *O* is the name of the invoked operation;
- *S* is the name of the service that provides *O*,
- *_Oid* is a variable whose value identifies the exact instance of *O*'s invocation within an operational SBS system session, and
- $\_P_1, \ldots, \_P_n$ are variables that indicate the values of the input parameters of O at the time of its invocation.

(ii) *Service operation reply events* – These are events that signify the return from the execution of an operation that has been invoked by the composition process of an SBS system in one of its partner services. These events are represented by terms of the form:

ir: S:O (_Oid)

- *ir* signifies this is a service operation reply event.
- *O, S, _Oid,* are as defined in (i).

Note that the values of the output parameters of such operations (if any) are represented by fluents which are initiated by the above event and it has been explained in Section 5.2.2.2 in Chapter 5.

(iii) *SBS operation invocation events* – These are events that signify the invocation of an operation in the composition process of an SBS system by one of its partner services. These events are represented by terms of the form:

rc:S:O (_Oid)

- *rc* signifies this is a SBS operation invocation event;
- *O, S,* and *O,_Oid* are as defined in (i).

Note that, the values of the input parameters of such operations (if any) are represented by fluents which are initiated by the above event and it has been explained in Section 5.2.2.2 in Chapter 5.

(iv) *SBS operation reply events* – These are events that signify the reply following the execution of an operation that was invoked by a partner service in the composition process of an SBS. These events are represented by terms of the form:

$re{:}S{:}O\ (\_Oid, \_P_1, \ldots, \_P_n)$

where

- *re* signifies this is a SBS operation reply event;
- $\_P_1, \ldots, \_P_n$ are variables that indicate the values of the output parameters of *O* at the time of its return, and
- *O, S, and _Oid* are as defined in (i).

(v) *Assignment events:* These are the events that signify the assignment of a value to a variable used in the composition process of an SBS system. These events are represented by terms of the form,

as:aname(_aid)

- *as* signifies this is an assignment event;
- *aname* is the name of the assignment in the composition process specification;
- *_aid* is a variable whose value identifies the exact instance of the assignment within an operational system session.

An assignment event initiates a fluent that represents the value of the relevant variable as discussed in Section 5.2.2.2 in Chapter 5.

3.3.4 Formulas

Figure 3.3 presents the logical syntax of formula in our framework in Extended Backus Naur Form (EBNF) [Sco93] [Iso96]. This definition defines the formal structure of a formula and does not define the terminal symbols. The precise definition of a formula in our framework is defined using XML schema [Xml04a]. A description of the schema is given in Section 3.3.5.

```
formula ::= {quantifiedTimeVariable}{quantifiedVariable}[body]head
body ::= logicalExpression
head ::= logicalExpression
logicalExpression ::= predicate | relationalPredicate
                    {(logicalOperator (predicate |
                     relationalPredicate | timePredicate))}
relationalPredicate ::= relationalOperand relationalOperator
                        relationalOperand
timePredicate ::= timeVariable relationalOperator timeVariable
relationalOperand ::= variable | operationCall | constantValue
relationalOperator ::= "<" | ">" | "<=" | ">=" | "=" | "!="
logicalOperator ::= "∧" | "∨"
```

Figure 3.3: The formal definition of a formula in EBNF

A **formula** comprises the following major parts,

Body and **Head:** A formula must have a head, which signifies the consequence (implication) of the formula. A formula may have a body, which signifies the antecedent (condition) of the formula. Both the body and the head must have a predicate or a relational predicate, which

may be followed by zero or any number of relational predicate or time predicate or predicate, each separated by a logical operator.

Logical Operator: A logical operator in a formula enables the logical combination of predicate, relational predicate and time predicates. A logical operator can be a conjunctive logical operator ($\wedge$) or a disjunctive logical operator ($\vee$).

Predicate: A predicate in a formula can be any of the predicates introduced in Section 3.3.1. It is discussed in Section 3.3.1 that predicates comprise events, fluents and time variables. In our framework, quantification for all the time variables appear in a predicate (or in the formula) should be explicitly specified. If the predicate represents a *Happens* predicate as it is introduced in Section 3.3.1, the boundaries for the time ranges $\Re$(LB, UB) in the predicate must be specified. If the variable *t* in such predicates is existentially quantified, at least one of LB and UB must be specified by using: (i) constant time indicators, or (ii) arithmetic expressions of time variables *t'* which appear in other *Happens* predicates of the same formula provided that the latter variables are universally quantified, and that *t* appears in their scope. If *t* is a universally quantified variable both LB and UB must be specified. *Happens* predicates with unrestricted universally quantified time variables take the form **Happens**(e,t,$\Re$(t,t)). These predicates express instantaneous events and are denoted as unconstrained predicate. The *Happens* predicates for which the time ranges $\Re$(LB, UB) have been specified (as described above) are denoted as constrained predicates.

All the non time variables appear in a predicate (or in the formula) are typed variable. In case of internal variable the type is defined in the specification (i.e. WSDL and BPEL file) of the SBS system. In case of external variable the user defines the type of the variable. More over non time variables should be quantified explicitly. If the quantifier for a non time variable is not specified it is assumed to be universally quantified.

Time Predicate: Time predicates are used to express time conditions between time variables in a formula by using the standard relational operators. A time variable in a formula signifies a time instance. For example, the literal $t1 < t2$ is true if *t1* is a time variable that signifies a time instance that occurred before a time instance signified by the time variable *t2*, and the literal $t1 = t2$ is true if *t1* is a time variable that signifies the same time instance as signified by the time variable *t2*.

Relational Predicate: Relational predicates are used to enable comparison among values of non time variables, return values of operation calls and constant values by using standard relational operators. More specifically,

- to compare the values of two different non time variables in a formula, e.g. *var1 > var2*
- to compare the return values of two different operation calls, e.g. *oc:self:avg(list) < oc:self:sub(val1, val2)*
- to compare the value of a non time variable with the return value of an operation call, e.g. *var1 < oc:self:add(val1, val2)*
- to compare the value of a non time variable with a constant value, e.g. *var1 < 100*
- to compare the return value of an operation call with a constant value, e.g. *oc:self:avg(list) < 100*

As shown in Figure 3.3, both the relational predicates and the time predicates are expressed using standard relational operators, these relational expressions in our framework must also be compliant with the basic axioms of equality and inequality [Bob01].

3.3.5 Formula Specification in XML

In Section 3.3.4 we presented the logic-based syntax of a formula in our monitoring framework. Our framework also supports specification of a formula in XML and this was motivated from the consideration that specification of formula in XML would increase the applicability of the framework. We have defined an XML schema to specify logic based formulas in XML. A graphical view of the schema is shown in Figure 3.4. Table 3.3 presents textual description of the key elements of the schema and the complete schema is presented in Appendix A.

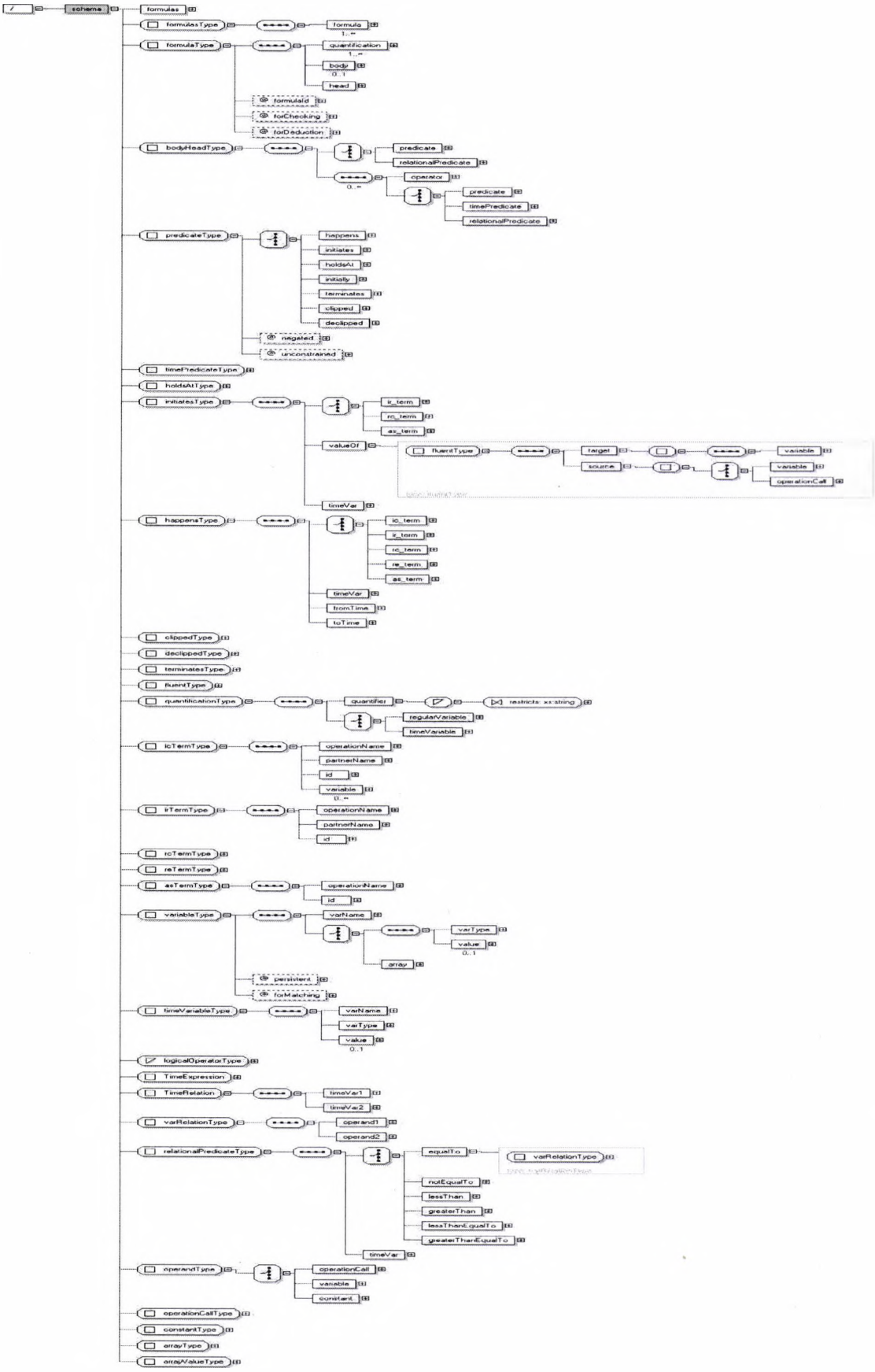


Figure 3.4: Graphical view of the formula schema

Table 3.3. Textual description of key elements of the formula Schema

Element	Description
<xs:element name="formulas" type="formulasType"/>	This is the element that would be used to define all the formulas in the XML document. It has type *formulasType*.
<xs:complexType name="formulasType"> <xs:sequence> <xs:element name="formula" type="formulaType" minOccurs="1" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType>	This element defines a list of formulas. It has one or more child element of type *formulaType* named as *formula*.
<xs:complexType name="formulaType"> <xs:sequence> <xs:element name="quantification" type="quantificationType" minOccurs="1" maxOccurs="unbounded"/> <xs:element name="body" type="bodyHeadType" minOccurs="0"/> <xs:element name="head" type="bodyHeadType"/> </xs:sequence> <xs:attribute name="formulaId" type="xs:string" use="required"/> </xs:complexType>	This element defines the structure of a formula. It has a required attribute *formulaId*. It has child elements in the following order: (i) at least one child element of type *quantificationType* named as *quantification* which is used to describe quantification of variables. (ii) zero or one child element of type *bodyHeadType* named as *body* which is used to describe the body part of the formula. (iii) one child element of type *bodyHeadType* named as *head* which is used to describe the head part of a formula.
<xs:complexType name="bodyHeadType"> <xs:sequence> <xs:element name="predicate" type="predicateType"/> <xs:sequence minOccurs="0" maxOccurs="unbounded"> <xs:element name="operator" type="operatorType"/> <xs:choice> <xs:element name="predicate" type="predicateType"/> <xs:element name="timePredicate" type="timePredicateType"/> </xs:choice> </xs:sequence> </xs:sequence> </xs:complexType>	This element defines the structure of head or body of a formula. It has a child element of type *predicateType* named as *predicate* which is used to define a predicate in the formula. *predicate* is followed by zero or more times (i) an element of type *operatorType* which is used to describe a logical operator (ii) an element of type *predicateType* named as *predicate* or an element of type *timePredicateType* named as *timePredicate*.
<xs:complexType name="predicateType"> <xs:choice> <xs:element name="happens" type="happensType"/> <xs:element name="initiates" type="initiatesType"/> <xs:element name="holdsAt" type="holdsAtType"/> <xs:element name="terminates" type="terminatesType"/> <xs:element name="clipped" type="clippedType"/> <xs:element name="declipped" type="declippedType"/> </xs:choice>	This element defines a predicate in a formula. *predicateType* has two attributes (i) *negated* which holds if the predicate is negated or not with default value set to *false*. (ii) *unconstrained* which holds if the predicate is unconstrained or not with default value is set to *false*. This element has only one child which could be any of the following types: (i) *happensType* named as *happens* which is used to represent EC *happens* predicate (ii) *initiatesType* named as *initiates* which is used to represent EC *initiates* predicate (iii) *holdsAtType* named as *holdsAt* which is used to represent EC *holdsAt* predicate (iv) *terminatesType* named as *terminates* which is used to represent EC *terminates* predicate (v) *clippedType* named as

<xs:attribute name="negated" type="xs:boolean" default="false"/> </xs:complexType>	*clipped* which is used to represent EC *clipped* predicate (vi) *declippedType* named as *declipped* which is used to represent EC *declipped* predicate
<xs:complexType name="happensType"> <xs:sequence> <xs:choice> <xs:element name="ic_term "type="icTermType"/> <xs:element name="ir_term" type="irTermType"/> <xs:element name="rc_term "type="rcTermType"/> <xs:element name="re_term "type="reTermType"/> <xs:element name="as_term" type="asTermType"/> </xs:choice> <xs:element name="timeVar" type="timeVariableType"/> <xs:element name="fromTime" type="TimeExpression"/> <xs:element name="toTime" type="TimeExpression"/> </xs:sequence> </xs:complexType>	This element is used to describe *happens* predicate of EC formula. It has four child elements in the following order: (i) a child element of type *irTermType* named as *ir_term* or of type *rcTermType* named as *rc_term* or of type *asTermType* named as *as_term* or of type *ictermType* named as *ic_term* or of type *reTermType* named as *re_term*. This child element is used to represent the event in the *happens* predicate. (ii) a child element of type *timeVariableType* named as *timeVar* which is used to represent time. (iii) a child element of type *TimeExpression* named as *fromTime* which is used to represent the starting time of a time range. (iv) a child element of type *TimeExpression* named as *toTime* which is used to represent the finishing time of a time range.
<xs:complexType name="rcTermType"> <xs:sequence> <xs:element name="operationName" type="xs:string"/> <xs:element name="partnerName" type="xs:string"/> <xs:element name="id" type=" xs:string "/> </xs:sequence> </xs:complexType>	This element is used to describe an event that signifies the receipt of an invocation of an operation from a partner service. It has three child elements in the following order: (i) a child element of type string named as *operationName* which represents the name of the operation. (ii) a child element of type string named as *partnerName* which represents the name of the partner service the operation belongs to. (iii) a child element of type string named as *id* which holds the id of the event.
<xs:complexType name="timePredicateType"> <xs:choice> <xs:element name="TimeEqualTo" type="TimeRelation"/> <xs:element name="TimeLessThan" type="TimeRelation"/> <xs:element name="TimeGreaterThan" type="TimeRelation"/> <xs:element name="TimeLessThanEqualTo" type="TimeRelation"/> <xs:element name="TimeGreaterThanEqualTo" type="TimeRelation"/> </xs:choice> </xs:complexType>	This element is used to express relation between two time values in the formula. It has one child element of type *TimeRealtion* named as any of the followings: (i) *TimeEqualTo* (ii) *TimeLessThan* (iii) *TimeGreaterThan* (iv) *TimeLessThanEqualTo* (v) *TimeGreaterThanEqualTo*
<xs:simpleType name="operatorType"> <xs:restriction base="xs:string"> <xs:pattern value="and|or"/> </xs:restriction> </xs:simpleType>	This element is used to define a logical operator. It is of type string and can have one of two possible values, *and* or *or*.
<xs:complxType name="operationCallType"> <xs:sequence>	This element is used define call to external or internal operations. . It has child elements in the

<xs:element name="name" type="xs:string"/> <xs:element name="partner" type="xs:string" minOccurs="0"/> <xs:element name="variable" type="variableType" minOccurs="0" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType>	following order: (i) a child element of type string named as *name* which represents the name of the operation, e.g. sub, add etc. (ii) an optional child element of type string named as *partner* which represents the name of the partner service the operation belongs to. (iii) zero or more child elements of type *variableType* named as *variable*, where each *variable* represents a parameter of the operation.
<xs:complexType name="variableType"> <xs:sequence> <xs:element name="varName" type="xs:string"/> <xs:choice> <xs:sequence> <xs:element name="varType" type="xs:string"/> <xs:element name="value" type="xs:string" minOccurs="0"/> </xs:sequence> <xs:element name="array" type="arrayType"/> </xs:choice> </xs:sequence> </xs:complexType>	This element is used to define a non time variable. It has child elements in the following order: (i) a child element of type *string*, named as *varName* that specifies the name of the variable, either (ii) two child elements of type *string* named as *varitype* and *value* that specifies type and value of primitive type variable, or (iii) one child element of type *arrayType* named as *array* that signifies that the variable is an array.

3.3.6 Example Formula

Figure 3.5 demonstrates some valid formula in our framework. These formulas specify functional property and quality of service property of a BPEL process called *RateTrackerProcess*. This BPEL process allows a user to convert any amount from one country currency to another country currency. The process receives two country names (*currency1* and *currency2*) and the amount to be converted from the user and invokes a web service, called *CurrencyExchanger* to perform the conversion. The web service *CurrencyExchanger* provides the operation *getRateRequest* that receives two currency names and the amount to be converted and it returns the converted *value*. The complete specification of the *RateTrackerProcess* (i.e. the BPEL file and the WSDL files) is provided in Appendix F.

Formula 1, in Figure 3.5 expresses a functional property of the *CurrencyExchanger* web service that specifies, given two countries and a fixed amount *getRateRequest* should return the same converted value. As discussed in Section 3.3.4 this formula is composed of a *body* and a *head*. The body is composed of six *predicates* and three *relational predicates*, where predicates and relational predicates are combined by conjunctive *logical operator* ($\wedge$). The first three predicates (see predicates

Happens(in:getRateRequest(id1,currency21,currency11,number1),t1,$\mathfrak{N}$(t1,t1)), **Happens**(ir:getRateRequest(id1),t2,$\mathfrak{N}$(t1,t2)) and **Initiates**(ir:getRateRequest(id1), valueOf(value,value1),t2)) in the formula signify an invocation to the *getRateRequest* operation and the corresponding response from the *getRaterequest* operation. The next three predicates (see predicates **Happens**(in:getRateRequest(id2,currency22,currency12,number2,t3,$\mathfrak{N}$(t2,t3)), **Happens**(ir:getRateRequest(id2),t4,$\mathfrak{N}$(t3,t4)) and **Initiates**(ir:getRateRequest(id2), valueOf(value,value2), t4)) signify another invocation to the *getRateRequest* operation and the corresponding response from the *getRaterequest* operation.. The three relational predicates (see relational predicates currecncy21 = currency22, currecncy11 = currency12 and number1 = number2) compare the values of the input variables of these two invocations. The head of the formula is composed of a single *relational predicate* (see value1 = value2) that compares the values returned by the *getRateRequest* operation for the two invocations made in the body of the formula. The quantifications for all the time variables appear in the formula are specified explicitly at the beginning of the formula.

Formula 1: (∀ t1: Time) (∃ t2, t3, t4: Time)
Happens(in:getRateRequest(id1,currency21,currency11,number1),t1,$\mathfrak{N}$(t1,t1)) ∧
Happens(ir:getRateRequest(id1),t2,$\mathfrak{N}$(t1,t2)) ∧
Initiates(ir:getRateRequest(id1), valueOf(value,value1),t2) ∧
Happens(in:getRateRequest(id2,currency22,currency12,number2,t3,$\mathfrak{N}$(t2,t3)) ∧
Happens(ir:getRateRequest(id2),t4,$\mathfrak{N}$(t3,t4)) ∧
Initiates(ir:getRateRequest(id2), valueOf(value,value2), t4) ∧
currecncy21 = currency22 ∧ currecncy11 = currency12 ∧ number1 = number2 ⇒ value1 = value2

Formula 2: (∀ t1: Time) (∃ t2: Time)
Happens(in:getRateRequest(id, currency2,currency1,number),t1,$\mathfrak{N}$(t1,t1)) ∧
Happens(ir:getRateRequest(id),t2,$\mathfrak{N}$(t1,t2)) ⇒ oc:self:sub(t2,t1) < Vo

Figure 3.5: Example formula in logic based syntax

Formula 2, in Figure 3.5 expresses a quality of service property of the *CurrencyExchanger* web service that specifies, the response time of the *getRateRequest* operation should be less than some predefined constant value. This formula is also composed of a *body* and a *head*. The body is composed of two *predicates*, where predicates are combined by conjunctive *logical operator* (∧). The first predicate (see predicate **Happens**(in:getRateRequest(id1,currency21,currency11,number1),t1,$\mathfrak{N}$(t1,t1)),) in the formula signify an invocation to the *getRateRequest* operation. The next predicates (see predicate **Happens**(ir:getRateRequest(id2),t1,$\mathfrak{N}$(t1,t2))) signify the corresponding response from the *getRaterequest* operation. The head of the formula is composed of a single *relational predicate* (see oc:self:sub(t2,t1) < Vo). This relational predicate involves an internal operation all, where the operation computes the response time of the *gerRateResponse* operation and the computed value is compared to a constant value. The quantifications for all the time variables appear in the formula are specified explicitly at the beginning of the formula.

The XML representation of Formula 2, is shown in Figure 3.6.

```
<formulas xmlns="http://tempuri.org/ec/formula">
  <formula formulaId="Formula_2" forChecking="true">
    <quantification>
      <quantifier>forAll</quantifier>
      <timeVariable>
        <varName>t1</varName>
        <varType>TimeVariable</varType>
      </timeVariable>
    </quantification>
    <quantification>
      <quantifier>existential</quantifier>
      <timeVariable>
        <varName>t2</varName>
        <varType>TimeVariable</varType>
      </timeVariable>
    </quantification>
    <body>
      <predicate negated="false"
                 unconstrained="true">
        <happens>
          <ic_term>
            <operationName>getRateRequest
                          </operationName>
            <partnerName>CurrencyExchanger
                          </partnerName>
            <id>vID1</id>
            <variable persistent="false"
                      forMatching="true">
              <varName>currency2</varName>
              <varType>string</varType>
            </variable>
            <variable persistent="false"
                      forMatching="true">
              <varName>currency1</varName>
              <varType>string</varType>
            </variable>
            <variable persistent="false"
                      forMatching="true">
              <varName>number</varName>
              <varType>int</varType>
            </variable>
          </ic_term>
          <timeVar>
            <varName>t1</varName>
            <varType>TimeVariable</varType>
          </timeVar>
          <fromTime>
            <time>
              <varName>t1</varName>
              <varType>TimeVariable</varType>
            </time>
          </fromTime>
          <toTime>
            <time>
              <varName>t1</varName>
              <varType>TimeVariable</varType>
            </time>
          </toTime>
        </happens>
      </predicate>
      <operator>and</operator>
      <predicate negated="false" unconstrained="false">
        <happens>
          <ir_term>
            <operationName>getRateRequest
                          </operationName>
            <partnerName>CurrencyExchanger
                          </partnerName>
            <id>vID2</id>
          </ir_term>
          <timeVar>
            <varName>t2</varName>
            <varType>TimeVariable</varType>
          </timeVar>
          <fromTime>
            <time>
              <varName>t1</varName>
              <varType>TimeVariable
                          </varType>
            </time>
          </fromTime>
          <toTime>
            <time>
              <varName>t2</varName>
              <varType>TimeVariable
                          </varType>
            </time>
          </toTime>
        </happens>
      </predicate>
    </body>
    <head>
      <relationalPredicate>
        <lessThan>
          <operand1>
            <operationCall>
              <name>sub</name>
              <partner>self</partner>
              <variable persistent="false"
                        forMatching="false">
                <varName>t2</varName>
                <varType>long</varType>
              </variable>
              <variable persistent="false"
                        forMatching="false">
                <varName>t1</varName>
                <varType>long</varType>
              </variable>
            </operationCall>
          </operand1>
          <operand2>
            <constant>
              <name>Vo</name>
              <value>1000</value>
            </constant>
          </operand2>
        </lessThan>
        <timeVar>
          <varName>t2</varName>
          <varType>TimeVariable</varType>
        </timeVar>
      </relationalPredicate>
    </head>
  </formula>
</formulas>
```

Figure 3.6: Example formula in XML

Chapter Four

Property Deviations and the Monitoring Scheme

4.1 Overview

In Chapter 2 of this thesis, we discussed the basic shortcomings of existing software requirement monitoring techniques in handling the monitoring of properties for service based systems and argued about the need for introducing types of deviations from monitoring properties that go beyond classical inconsistency and developing reasoning mechanisms to detect them.

In this chapter, we introduce the types of property deviations that are important to detect in service based systems and define the algorithms underpinning the reasoning mechanism that detect them. We also introduce an example of a service based system that is used to explain new deviation types.

4.2 An Example of a Service Based System

To demonstrate the need for the types of property deviations which may arise in service based systems, and clarify their meaning we use an example of a service based system that will be referred to as *Car Rental System* (or shortly "CRS") in the rest of this thesis. The structure of the Car Rental System is shown in Figure 4.1.

This system acts as a broker offering its customers the ability to rent cars provided by different car rental companies directly from car parks at different locations. CRS is implemented as a service composition process, which interacts with *Car Information Services* (IS), *Customer Management Service* (CMS), *User Interaction Services* (UI) and *Sensoring Services* (SS).

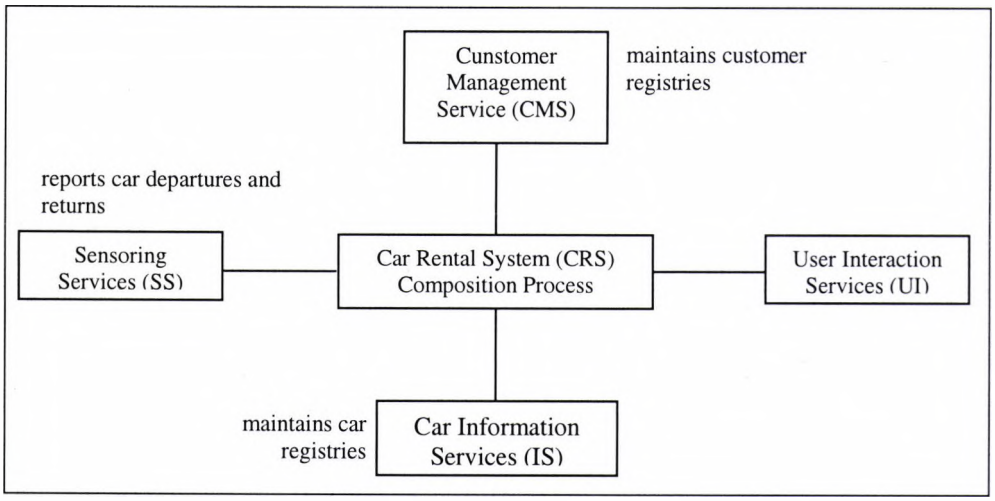


Figure 4.1: Structure of the car rental system (CRS)

These services realise the following functionalities within CRS:

- The *Car information services* (IS) are provided by different car rental companies to maintain registries of cars that can be rented, check car availability upon car rental requests and allocate cars to customers as requested by CRS.

- The *Sensoring services* (SS) are provided by different car parks to detect movements of cars as they are driven in or out of car parks and inform CRS accordingly.

- The *Customer Management Service (CMS)* maintains the database of the customers of CRS and authenticates these customers as requested by CRS.

- The *User interaction services* (UI) provide CRS with different user interfaces that handle interactions with the end-users.

Typically, CRS receives car rental requests from UI services, authorises customers contacting CMS and checks for the availability of cars by contacting IS services, and gets car movement information from SS services.

Figure 4.2 and Figure 4.3 show a partial specification of CRS service based system expressed in EC formulas. As we introduced in Chapter 2 the specification of a service based system includes four different types of formulas, namely *behavioural properties, functional properties, quality of service properties* and *assumptions.*

Figure 4.2 shows specifications of three *behavioural properties* of CRS service based system expressed in EC. As introduced in Chapter 2, behavioural properties alternative paths of the

service composition process and they signify properties with respect to the whole composition process. In the EC formulas that define these properties (and all the other EC formulas which are used in this Chapter),

- Non-time variables are assumed to be universally quantified and their name is preceded by an underscore (see _pID in the behavioural property BP1 in Figure 4.2 for example)
- Time variables are of the form t1,t2,…,tn and explicitly quantified
- t_u refers to the minimum time between the occurrences of two events and is specified by the system provider.

Behavioural properties:

(BP1) (∀ t1:Time) (∃t2:Time) (∃t3:Time)
Happens(rc:UI:CarRequest(_oID1),t1,ℜ(t1,t1)) ∧
Initiates(rc:UI:CarRequest(_oID1), valueOf(info.p,_pID),t1) ∧
Happens(in:IS:FindAvailableCar(_oID2, _pID),t2, ℜ(t1,t2)) ∧
Happens(ir:IS:FindAvailableCar(_oID2),t3, ℜ(t2,t3)) ∧ **Initiates**(ir:IS:FindAvailableCar(_oID2), valueOf(info.v,_vID),t3) ⇒ (∃t4:Time) **Happens**(re:UI:CarRequest(_oID3,_vID, _pID), t4, ℜ(t3,t3+3*t_u))

(BP2) (∀ t1:Time) (∃t2:Time)
Happens(re:UI:CarRequest(_oID1,_vID, _pID),t1, ℜ(t1,t1)) ∧ **Happens**(rc:SS:Depart(_oID2),t2, ℜ(t1,t1+6*t_u) ∧ **Initiates**(rc:SS:Depart(_oID2),valueOf(info.v, _vID),t2) ∧
Initiates(rc:SS:Depart(_oID2),valueOf(info.p, _pID),t2) ⇒
(∃t3:Time) **Happens**(in:IS:MakeUnAvailable(_oID3,_vID,_pID)),t3, ℜ(t1+7*t_u,t1+7*t_u))

(BP3) (∀ t1:Time) (∃t2:Time)
Happens(re:UI:CarRequest(_oID1,_vID, _pID),t1, ℜ(t1,t1)) ∧
¬ **Happens**(rc:SS:Depart(_oID2),t2, ℜ(t1,t1+6*t_u) ⇒
(∃t3:Time) **Happens**(in:IS:MakeAvailable(_oID3,_vID,_pID)),t3, ℜ(t1+7*t_u,t1+7*t_u))

Figure 4.2: Behavioural properties of the car rental system

The formulas in Figure 4.2 specify behavioural properties of CRS which as we discussed in Chapter 2 are automatically extracted from the specification of the composition process of it that is expressed in BPEL:

- The formula *BP1* specifies that when CRS receives a request for renting a car from a UI service (see the predicate **Happens**(rc:UI:CarRequest(_oID1),t1,ℜ(t1,t1)) in the formula), it will invoke the operation *FindAvailableCar* in the IS service to search for available cars (as specified by the predicate **Happens**(in:IS:FindAvailableCar(_oID2,_pID),t2,ℜ(t1,t2))) and will report any cars which are identified by this operation back to the UI service within 3 time units (as specified by the predicate **Happens**(re:UI:CarRequest(_oID3,_vID),t4, ℜ(t3,t3+t_u)) in the formula) it receives a response from the IS service.

- The formula *BP2* specifies that when CRS replies to a car request made by UI service that signifies the release of a car key to a customer (see the predicate **Happens**(re:UI:CarRequest(_oID1,_vID, _pID),t1, $\Re$(t1,t1)) in the formula), it waits for an event signifying the exit of the car from the car park for 6 time units (this message is to be sent by SS, see the predicate **Happens**(rc:SS:Depart(_oID2),t2, $\Re$(t1,t1+6*t_u) in the formula). If the latter event occurs, CRS invokes the IS to mark the relevant car as unavailable as specified by the predicate **Happens**(in:IS:MakeUnAvailable(_oID3,_vID,_pID)),t3, $\Re$(t1+7*t_u,t1+7*t_u)) in the formula.

- The formula *BP3* specifies that when CRS replies to a car request made by UI service that signifies the release of a car key to a customer (see the predicate **Happens**(re:UI:CarRequest(_oID1,_vID, _pID),t1, $\Re$(t1,t1)) in the formula), it waits for an event signifying the exit of the car from the car park for 6 time units (this message is to be sent by SS, see the predicate **Happens**(rc:SS:Depart(_oID2),t2, $\Re$(t1,t1+6*t_u) in the formula). If the latter event does not occur, CRS invokes the IS to mark the relevant car as available as specified by the predicate **Happens**(in:IS:MakeAvailable(_oID3,_vID,_pID)),t3, $\Re$(t1+7*t_u,t1+7*t_u)) in the formula.

In Figure 4.3, we also show EC formulas that specify some of the *functional properties*, *quality of service properties* and *assumptions* of the CRS. As discussed in Chapter 2, functional properties signify functional requirement of a service of group of services deployed by the composition process. Also, as it may be recalled from Chapter 2, assumptions are used to generate additional information about the expected service behaviour and its effect on the state of the system. More specifically in this example,

- The formula *FP1* specifies a functional property regarding the correctness of the behaviour of the sensoring services used by CRS. According to it, if a car is sensed to enter a car park at some time *t1* as reported by the call of the operation *Enter* in *CRS* by a sensoring service SS (see the predicate **Happens**(rc:SS:Enter(_oID1),t1,$\Re$(t1,t1))) and later at some time *t2* the same car is sensed to enter in the same or a different car park (see the predicate **Happens**(rc:SS:Enter(_oID2),t2,$\Re$(t1+t_u,t2))), then the departure of the relevant car from the first car park must have also been reported to CRS by SS between the two events that notify the entrance of the car in the car parks (see the predicate **Happens**(rc:SS:Depart(_oID3),t3,$\Re$(t1+t_u,t2-t_u))).

Functional properties:

(FP1) (∀ t1:Time) (∃ t2:Time) **Happens**(rc:SS:Enter(_oID1),t1,ℜ(t1,t1)) ∧
Initiates(rc:SS:Enter(_oID1), valueOf(info.v, _vID), t1) ∧
Initiates(rc:SS:Enter(_oID1), valueOf(info.p, _pID1), t1) ∧
Happens(rc:SS:Enter(_oID2),t2,ℜ(t1+t_u,t2)) ∧
Initiates(rc:SS:Enter(_oID2), valueOf(info.v, _vID), t2) ∧
Initiates(rc:SS:Enter(_oID2), valueOf(info.p, _pID2), t2) ⇒ (∃t3:Time)
Happens(rc:SS:Depart(_oID3),t3,ℜ(t1+t_u,t2−t_u)) ∧
Initiates(rc:SS:Depart(_oID3), valueOf(info.v, _vID), t3) ∧
Initiates(rc:SS:Depart(_oID3), valueOf(info.p, _pID1), t3)

(FP2)(∀ t1:Time) (∃ t2:Time) **Happens**(in:IS:FindAvailableCar(_oID,_pID), t1, ℜ(t1,t1)) ∧
Happens(ir:IS:FindAvailableCar(_oID), t2, ℜ(t1,t2)) ∧ (∃ _c:Car)
HoldsAt(valueOf(_c.carID, _vID), t2−t_u) ∧
HoldsAt(valueOf(_c.availability, "not available"), t2−t_u) ⇒
¬**Initiates**(ir:IS:FindAvailableCar (_oID), valueOf(info.v, _vID), t2)

(FP3) (∀ t1:Time) (∃ t2, t3:Time) (∃ _c:Car) **Happens**(in:UI:RelKey(_oID1,_vID), t1,ℜ(t1, t1)) ∧
Happens(ir:UI:RelKey(_oID1), t2, ℜ(t1, t2)) ∧
Happens(rc:UI:RetKey(_oID2),t3,ℜ(t2,t3)) ∧
Initiates(rc:UI:RetKey(ID), valueOf(info.v, _vID),t3) ∧
Initiates(rc:UI:RetKey(ID), valueOf(info.p, _pID),t3) ∧ (t2 ≤ t4) ∧ (t4 ≤ t3)⇒
HoldsAt(valueOf(_c.carID, _vID), t4) ∧
HoldsAt(valueOf(_c.availability, "not available"),t4)

(FP4) (∀ t1:Time)**Happens**(rc:SS:Enter(_oID1),t1,ℜ(t1,t1)) ∧
Initiates(rc:SS:Enter(_oID1),valueOf(info.v,_vID),t1) ∧
Initiates(rc:SS:Enter(_oID1),valueOf(info.p,_pID),t1) ⇒ (∃t2:Time)
Happens(rc:UI:RetKey(_oID2),t2, ℜ(t1+t_u,t1+10*t_u)) ∧
Initiates(rc:UI:RetKey(_oID2),valueOf(info.v, _vID),*t2)* ∧
Initiates(rc:UI:RetKey(oID2),valueOf(info.p, _pID),t2)

Quality of service properties:

(QP1)(∀ t1:Time) (∃t2:Time) **Happens**(in:IS:FindAvailableCar(_oID1, _pID),t1, ℜ(t1,t1)) ∧
Happens(ir:IS:FindAvailableCar(_oID1),t2, ℜ(t1,t2)) ⇒ oc:self:sub(t2, t1) < 1

(QP2) (∀ t:Time) (t ≤ 0) ∧ (t ≥ 86400) ∧ (∃ _x: list of Real) **HoldsAt**(valueOf(FindAvailableCarRT, _ x), t)
⇒ oc:self:avg(_x) < 1

Assumptions:

(A1) (FindAvailableCarRT: list of Real) (∀ t1:Time) (∃t2:Time)
Happens(in:IS:FindAvailableCar(_oID1, _pID),t1, ℜ(T_s, T_e)) ∧
Happens(ir:IS:FindAvailableCar(_oID1),t2, ℜ(t1,t2)) ∧
HoldsAT(valueOf(FindAvailableCarRT, _ x), t2) ⇒
Initiates(ir:IS:FindAvailableCar(_oID1), valueOf(FindAvailableCarRT, oc:self:append(_x,
oc:self:sub(t2, t1)), t2)

(A2) (∀ t1:Time) (∃ t2:Time) **Happens**(in:UI:RelKey(_oID1,_vID), t1,ℜ(t1, t1)) ∧
Happens(ir:UI:RelKey(_oID1), t2, ℜ(t1, t2)) ∧ (∃ _c:Car) **HoldsAt**(valueOf(_c.carID, _vID), t2)
⇒ **Initiates**(ir:UI:RelKey(_oID1), valueOf(_c.availability, "not available"), t2 + t_u)

(A3) (∀ t1:Time) (∃ t2:Time) **Happens**(in:UI:RelKey(_oID1,_vID), t1,ℜ(t1, t1)) ∧ ¬(∃ _c:Car)
HoldsAt(valueOf(_c.carID, _vID), t1) ⇒
Initiates(in:UI:RelKey(_oID1), valueOf(oc:self:new(Car).carID,_vID, t1)

(A4) (∀ t1:Time) (∃ _c:Car) **Happens**(rc:UI:RetKey(_oID1), t1,ℜ(t1, t1)) ∧
Initiates(rc:UI:RetKey(_oID1), valueOf(info.v, _vID), t1) ∧ **HoldsAt**(valueOf(_c.carID, _vID), t1) ⇒
Initiates(ir:UI:RelKey(_oID1), valueOf(_c.availability, "available"), t1 + t_u)

(A5) (∀ t1:Time) **Happens**(rc:UI:RetKey(_oID1), t1,ℜ(t1, t1)) ∧
Initiates(rc:UI:RetKey(_oID1), valueOf(info.v, _vID), t1) ∧
¬(∃ _c:Car) **HoldsAt**(valueOf(_c.carID, _vID), t1) ⇒
Initiates(rc:UI:RetKey(_oID1), valueOf(oc:self:new(Car).carID,_vID, t1)

Figure 4.3: Functional properties, assumptions and QoS properties of the car rental system

- The formula *FP2* in Figure 4.3 expresses a functional property for the operation *FindAvailableCar* of the IS service. According to this requirement, *FindAvailableCar* should not return the identifier of a car that is not considered to be available. In this formula, car availability is indicated by the user defined state variable (i.e. a variable that is not declared in the composition process of the system) that has been introduced to record car availability and represent the effect of different system events on to it. The value of this variable is determined by events that signify the release of car keys to customers and the returns of these keys as defined by the assumptions *A2–A5*. More specifically, according to *A2*, the release of the key of a car will make this car unavailable and according to *A4*, the return of the key of a car will make this car available again. The formulas *A3* and *A5* are used to create a special object to represent the availability status of a specific car within the monitoring process.

  A new object is created to represent this availability only if no such object has been created before. It should be noted that the availability status of the car as recorded by this variable may be different from the availability status of the same car as recorded by the IS service of CRS. This is because the monitor has its own knowledge about the state of the system that is determined by the assumptions made and which is not necessarily the same as the real system state.

- The formula *FP3* specifies a functional property regarding the availability status of a car. According to this, if a car key is released at a time point (see the predicates **Happens**(in:UI:RelKey(_oID1,_vID), t1,ℜ(t1, t1)) and **Happens**(ir:UI:RelKey(_oID1), t2, ℜ(t1, t2))) and the same car key is returned at some time point later, (see the predicates **Happens**(rc:UI:RetKey(_oID2),t3,ℜ(t2,t3)),**Happens**(rc:UI:RetKey(_oID2),t3, ℜ(t2,t3)) and **Initiates**(rc:UI:RetKey(ID), valueOf(info.p, _pID),t3)) then the car is not available between these two time points (see the predicates **HoldsAt**(valueOf(_c.carID, _vID), t4) and **HoldsAt**(valueOf(_c.availability, "not available"),t4)).

- The formula *FP4* specifies a functional property that signifies if a car enters a car park (see the predicate **Happens**(rc:SS:Enter(_oID1),t1,ℜ(t1,t1))), the key of the car should be returned within the next 10 time units (see the predicate **Happens**(rc:UI:RetKey(_oID2),t2, ℜ(t1+t_u,t1+10*t_u))).

Figure 4.3 also shows examples of quality of service properties expressed in EC. As introduced in Chapter 2, quality of service properties signifies quality requirements of the composition process or a service/group of services deployed by the composition process. More specifically,

- The formula *QP1* specifies that the response time of the operation *FindAvailableCar* of the IS services should be always less that 1 second (assuming that time is recorded in seconds). It should be noted that this formula calls the internal operation *sub(t2,t1)* of the monitor to compute the response time of *FindAvailableCar* (see the term oc:self:sub(t2,t1) in the relational predicate of the formula).

- The formula *QP2* specifies that the average response time of the operation *FindAvailableCar* of the IS services at any point in the first hour of the operation of a system (i.e., the period from 0 to 86400 assuming that time is recorded in seconds) should be less that 1 seconds. This formula calls the internal operation *avg(a)*. This operation calculates the average response time of *FindAvailableCar* using the recorded response times of this operation stored in the list *FindAvailableCarRT* (*FindAvailableCarRT* is a user defined variable). The recording of the response times of the operation FindAvailableCar in the list *FindAvailableCarRT* is performed by the formula A1 that is an assumption used to initialize *FindAvailableCarRT* to a new list of recorded times that includes the newly observed response time.

4.3 Property Deviations

Assuming a set of behavioural properties B_S, a set of QoS properties Q_S, a set of functional properties F_S and a set of assumptions A_S all expressed in the restricted form of the event calculus that we introduced in Chapter 3, we define five different types of deviations from a requirements specification, $B_S \cup Q_S \cup F_S$, of a service-based system [Mah04][Spa04][Spa06]. These types of deviations are: (i) inconsistency of recorded behaviour; (ii) inconsistency of expected behaviour; (iii) unjustified behaviour; (iv) possible inconsistency of expected behaviour; and (v) potentially unjustified behaviour. These types of deviations are defined in terms of the *recorded* and *expected* behaviour of a system, which are defined as follows:

Definition 1: The *recorded behaviour* of a system S at time T, $E_R(T)$, is a set of event, and fluent initiation or termination literals of the forms: **Initally**(f), **Happens**(e,t,$\Re$(t,t)) and **Initiates**(e,f,t) that have been recorded during the operation of S and for which $0 \leq t$, $t \leq T$,

and the set of **HoldsAt**, **Terminates** literals that can be derived from them due to the axioms of EC.

The expected behaviour of a system with respect to a subset R_S of the functional properties and the assumptions of it includes the literals of its recorded behaviour and any other event or fluent initiation, termination and holding literals that can be derived from them and the formulas in R_S, or formally:

Definition 2: The *expected behaviour* of a system S given a subset R_S of its functional properties and assumptions ($R_S \subseteq (F_S \cup A_S)$) at time T is a set of events $E_U(R_S,T)$ that is defined as: $E_U(R_S,T) = \{e \mid (f \in R_S)$ and $\{E_R(T),f\} \models_{nf} e\}$, where $\models_{nf}$ signifies entailment using the normal rules of inference of first-order logic and the principle of *negation as failure* [Cla78].

According to Definition 2, the unrecorded expected behaviour of a system at time T given a set of formulas R_S includes events, which can be derived from the formulas in R_S.

4.3.1 Inconsistency of Recorded Behaviour

An inconsistency between the recorded behaviour of a system S and a functional property or QoS property of S at time T is defined as follows:

Definition 3: A functional property or a QoS property f of the form $f: C \Rightarrow A$ is *inconsistent* with the recorded behaviour of a system S until time t if and only if: $\{E_R(t)\} \models_{nf} \neg f$, where $\models_{nf}$ signifies entailment using the normal rules of inference of first-order logic and the principle of negation as failure.

Definition 3 establishes a classic notion of inconsistency in system behaviour that can be detected by checking whether the negation of a formula is entailed by the recorded run-time behaviour of a system. To this end, it is equivalent to the notion of inconsistency discussed in [Fea98]. It should be appreciated that inconsistency of recorded behaviour may only be detected with respect to QoS property or functional property, but not behavioural properties. This is because the latter are automatically extracted from the source code of the composition process and thus they represent definite sequences of events in the process execution. Therefore, they may be violated only if the process ceases to be alive. For example consider the behavioural property *BP2* in Figure 4.2, let CRS replies to a car request made by UI service and receives an event from SS that signifies the exit of the car from the car park within 6 time units, but CRS goes down before invoking the IS to mark the relevant car as

unavailable. In such a situation *BP2* will be detected as inconsistent with the recorded behaviour of the system.

Example. Suppose that the log of events of a system includes the literals shown in Figure 4.4[1]. Given this event log, the recorded behaviour of CRS is inconsistent with the functional property *FP1*, which is about the SS service of the system. This is because there are two events that signify the entrance of *veh1* first to car park *loc1* at T=1 (see literals L1-L3 in Figure 4.4) and, subsequently, to car park *loc3* at T=27 (see literals L4-L6 in Figure 4.4) but no depart event in between them to signify the departure of *veh1* from *loc1*. This inconsistency demonstrates a functional problem of the sensoring service at the car park *loc1*.

L1: **Happens**(rc:SS:Enter(op1),1,ℜ(1,1))
L2: **Initiates**(rc:SS:Enter(op1), valueOf(info.v,veh1),1)
L3: **Initiates**(rc:SS:Enter(op1), valueOf(info.p,loc1),1)
L4: **Happens**(rc:SS:Enter(op2),27,ℜ(27,27))
L5: **Initiates**(rc:SS:Enter(op2), valueOf(info.v,veh1),27)
L6: **Initiates**(rc:SS:Enter(op2), valueOf(info.p,loc3),27)
L7: **Happens**(in:UI:RelKey(op3, veh2),28, ℜ(28,28))
L8: **Happens**(ir:UI:RelKey(op3), 29, ℜ(29,29))
L9: **Happens**(rc:UI:CarRequest(op4),30, ℜ(30,30))
L10: **Initiates**(rc:UI:CarRequest(op4),valueOf(info.p,loc3),30)
L11: **Happens**(in:IS:FindAvailableCar(op5,loc3),31, ℜ(31,31))
L12: **Happens**(ir:IS:FindAvailableCar(op5),33, ℜ(33,33))
L13: **Initiates**(ir:IS:FindAvailableCar(op5), valueOf(info.v,veh1),33)
L14: **Happens**(re:UI:CarRequest(op6,veh1,loc3), 33, R(33,33))
L15: **Happens**(in:IS:MakeAvailabe(op7,veh1,loc3), 40, R(40,40))
L16: **Happens**(rc:UI:CarRequest(op8),41, ℜ(41,41))
L17: **Initiates**(rc:UI:CarRequest(op8),valueOf(info.p,loc2),41)
L18: **Happens**(in:IS:FindAvailableCar(op9,loc2),42, ℜ(42,42))
L19: **Happens**(ir:IS:FindAvailableCar(op10), 43, ℜ(43,43))
L20: **Initiates**(ir:IS:FindAvailableCar(op10), valueOf(info.v,veh2),43)
L21: **Happens**(re:UI:CarRequest(op11,veh2,loc2), 44, R(44,44))
L22: **Happens**(rc:UI:RetKey(op12),45,ℜ(45,45))
L23: **Initiates**(rc:UI:RetKey(op12), valueOf(info.v, veh1), 45)
L24: **Initiates**(rc:UI:RetKey(op12), valueOf(info.p, loc3), 45)
L25: **Happens**(rc:SS:Enter(op13),53,ℜ(53,53))
L26: **Initiates**(rc:SS:Enter(op13), valueOf(info.v,veh2),53)
L27: **Initiates**(rc:SS:Enter(op13), valueOf(info.p,loc4),53)
L28: **Happens**(rc:UI:RetKey(op14),54,ℜ(54,54))
L29: **Initiates**(rc:UI:RetKey(op14), valueOf(info.v, veh2), 54)
L30: **Initiates**(rc:UI:RetKey(op14), valueOf(info.p, loc4), 54)
L31:**Happens**(rc:UI:CarRequest(op15),55, ℜ(55,55))

Figure 4.4: Event log of car rental system.

[1] It should be noted that the runtime events (or monitoring events) presented in Figure 4.4 is different from the low level EC events introduced in Chapter 3. A runtime event (monitoring event) is denoted as an EC predicate that comprises a low level EC event or/and an EC fluent presented in Chapter 3 and a specific time point. For example the EC predicate **Happens**(rc:P1:receiveRequest(id3), 5, ℜ(5,5)) represents a runtime event (monitoring event) since it signifies an instance of a low level EC event (i.e. rc:P1:receiveRequest(id3)) at time point 5. In the rest of the thesis the word "event" is used to refer a runtime event (or monitoring event) with this semantic. The event receiver in our monitoring framework converts a low level EC event to a runtime event and this mechanism is described in Chapter 5.

QoS properties may also be inconsistent with the recorded behaviour of the system. For example the QoS property *QP1* in Figure 4.3 is violated at T=33 as the response time of the operation *FindAvailableCar* is 2 seconds (see literal L11 that signifies a call to the operation *FindAvailableCar* and the literal L12 that signifies the response from the operation *FindAvailableCar*).

4.3.2 Inconsistency of Expected Behaviour

An inconsistency between a behavioural property, a functional property or a QoS property of a system *S* and its expected behaviour is defined as follows:

Definition 4: A behavioural property, functional property or a QoS property of the form $f: C \Rightarrow A$ is inconsistent with the *expected behaviour* of a system *S* at time *T* if and only if: $\{E_R(T), E_U(dep(f) \cup EC_a, T), \} \models_{nf} \neg f$, where:

- $dep(f)$ is the set of formulas $F: B \Rightarrow H$ in $(F_S \cup A_S) - \{f\}$ which f depends on. A formula $F: B \Rightarrow H$ belongs to $dep(f)$, if its head H has a literal L that unifies with: (i) some literal K in the body C or in the head A of f, or (ii) some literal K in the body B'' or in the head H'' of another formula F'' that belongs to $dep(f)$; and

- EC_a are the axioms of event calculus (see Chapter 3)

According to this definition, the check about the inconsistency of a formula f with the expected behaviour of a system must, in addition to events which are recorded in $E_R(T)$, take into account events which should have been generated according to other formulas in $F_S \cup A_S$ and can affect the satisfiability of f. The definition of the set $dep(f)$ in Definition 4 is similar to the notion of direct and indirect dependency in [Ple93].

Examples. Given the recorded event log of Figure 4.4, the functional property *FP2* is violated by the expected behaviour of CRS. *FP2* is a functional property about the behaviour of *IS* services. According to this functional property, the operation *FindAvailableCar*, which is provided by the IS service of CRS and searches for available cars at specific car parks should not return the identifier of a car to CRS unless this car is available. The violation of *FP2* in this case occurs since from the functional property *FP3* in Figure 4.3 we can derive that *veh2* could not be available from T=30 when its key was released (see literals L7 and L8 in Figure 4.4) until T=53 (that is one time unit before its key was returned back - see literals L28, L29

and L30 in Figure 4.4). Nevertheless, the execution of the operation *FindAvailableCar* of the IS service at T=43 reported *veh2* as an available vehicle (see literal L20 in Figure 4.4).

Inconsistencies with respect to expected behaviour may also indicate violations of QoS properties in cases where the specification of these properties uses values that should be derived from the recorded behaviour of the system. For instance, the QoS property *QP2* in Figure 4.3 is violated at T=33 as the average response time of the operation *FindAvailableCar* is 2 seconds (i.e., one recorded call with response time of 2 seconds according to literals L11 and L12). This violation is caused by the expected behaviour of the system since *QP2* is evaluated by performing a calculation over the value of the fluent *FindAvailableCarRT* which is derived by assumption A1. The same QoS requirement is also violated at time T=43 when, following the second call of the operation *FindAvailableCar*, its average response time is 1.5 seconds.

4.3.3 Unjustified Behaviour

The third type of deviations that can be detected by our framework occurs when the conditions of a behavioural property f that has generated an event e are satisfied by the recorded system behaviour but violated by the expected system behaviour. In such cases, the generation of the event e is the result of wrong assumptions about the satisfiability of the conditions of f that the system has made at run-time, and constitutes what we refer to as "unjustified behaviour". This type of deviation is formally defined as follows:

Definition 5: A behavioural property of the form $f: C \Rightarrow A$ is said to generate *unjustified behaviour* if and only if there is an event e such that

(i) $e \in E_R(t)$

(ii) e can be unified with A

(iii) $\{E_R(t) - \{e\}, f, EC_a\} \models_{nf} e$

(iv) $\{E_R(t) - \{e\}, B_S - \{f\}, EC_a\} \not\models_{nf} e$ and

(v) there is a literal L in C for which, $\{E_R(t), E_U(dep(f),t), EC_a\} \models_{nf} \neg L$

The conditions (i)-(iv) in Definition 5 identify an event e which has been generated by the system due to the realisation of a formula f (condition (iv) guarantees that e cannot have been generated by some other formula). The satisfaction of conditions (i)-(iv) implies that the conditions of f are satisfied by the recorded behaviour of the system. Note, however, that according to condition (v), there is some condition in C that would not be satisfied if all the

events that could be generated by formulas which f depends on are taken into account. In such cases, e is the result of behaviour that is based on wrong assumptions about the satisfiability of the conditions of f that the system made at run-time.

Example. Given the event log of Figure 4.4, a case of unjustified behaviour of the CRS system that has been caused by the behavioural property *BP1* can be detected at T=54 (*BP1* states that following the receipt of a request for a car rental, CRS will contact IS services to find an available vehicle and if such a vehicle can be found it will reply to the request). More specifically in this case, as the literals L16–L20 in Figure 4.4 indicate, all the conditions of *BP1* were satisfied at T=43 and therefore CRS replied to the car hire request that it had received at *T=44* (see the literal L21 in Figure 4.4) as specified by *BP1*. Note, however, that if the IS and SS services of CRS had behaved according to the functional properties FP2 and FP3 respectively the condition *Initiates(ir:IS:FindAvailableCar(_oID2), valueOf(info.v,_vID),t3)* of *BP1* would have been violated. The violation of this condition of *BP1* can be deduced from:

- the literals L18 and L19 the event log of Figure 4.4;
- the functional property *FP2* about the behaviour of SS (*FP2* belongs to *dep(BP1)*), and
- the literals *HoldsAt(valueOf(_c.carID, veh2), 42)* and *HoldsAt(valueOf(_c.availability, "not available), 42)* that can be derived from the literals L7, L8, L28, L29 and L30 in the event log of Figure 4.4 and the functional property *FP3* (*FP3* belongs to *dep(FP2)*).

In other words, if IS and SS had behaved as expected by the functional properties *FP2* and *FP3* in this case, *veh2* should not have been reported by the operation *FindAvailableCar* as available and, subsequently, *veh2* should not have been hired.

4.3.4 Possible Inconsistency of Expected Behaviour

The fourth type of deviation captures potential violations of behavioural properties, functional properties or QoS properties of a system. The possibility of such violations is established due to events and fluents (states) which may have occurred during the operation of a service-based system without however having been recorded in the event log of it. More specifically, a possible violation of a formula f may arise if, in addition to the events which have been recorded during the operation of a system, we also take into account events that can be derived from by: (a) deductive reasoning using the formulas in $F_S \cup A_S$, (b) abductive

reasoning using the formulas in $F_S \cup A_S$, or (c) assuming the occurrence of events in time ranges that overlap with the time ranges in which these events are known to have occurred.

Definition 6: A behavioural property, a functional property or a QoS property of the form *f: C⇒A* is possibly violated at time *t* if and only if:

$\{E_R(t), E_U(dep(f),t), E_U(abd(f),t), EC_a, \alpha_{pHappens}\} \models \neg pForm(f)$

where

- *abd(f)* is a set of complement formulas in $F_S \cup A_S - \{f\}$ that *f* depends on (see Definition 4). The complement of a formula *F: B⇒H* in $F_S \cup A_S - \{f\}$, is included in *abd(f)* only if the following criteria is satisfied,

  (i) The head *H* of *F* includes at least one non-negated *Happens* predicate, which appears in at least one formula in B_S.

  (ii) The body *B* of *F* contains at least one predicate, which does not appear in the head of any other formula in $F_S \cup A_S - \{f\}$.

  The complement of a formula *F: B⇒H* in $F_S \cup A_S - \{f\}$ is defined as *F': H⇒B*. This process of complementing a formula reverses the time relations and the quantifiers of variables. For example, consider the following formula,

  (F1) $(\forall\ t1)\text{Happens(ic:S1:P(ID, x)},t1,\Re(t1,t1)) \Rightarrow (\exists\ t2)\text{Happens(ic:S1:Q(ID, y)},t2,\ \Re(t1+t_u,t1+10*t_u))$

  From this formula we have $t1+t_u \leq t2$ and $t2 \leq t1 + 10*t_u$, these relations can be rewritten as $t1 \leq t2 - t_u$ and $t2 - 10*t_u \leq t1$ respectively. In the reverse formula time range for P is defined in terms of the time variable of Q, so the quantifier of t2 would be changed to the universal quantifier and the quantifier of t1 would be changed to the existential quantifier. Therefore the reverse formula of formula F1 would be:

  $(\forall\ t2)\ \text{Happens(ic:S1:Q(ID, y)},t2,\ \Re(t2,t2)) \Rightarrow (\exists\ t1)\text{Happens(ic:S1:P(ID, x)},t1,\Re(\ t2-10*t_u,,t2-t_u))$

- *pForm(f)* is a formula that is produced from *f* if all the occurrences of the EC predicates in *f* are replaced by the corresponding *p-EC* predicates. The *p-EC* predicates for *Happens*, *Initiates, Terminates* and *HoldsAt* predicates are *pHappens, pInitiates, pTerminates* and *pHoldsAt* respectively. These *p-EC* predicates are defined as follows,

  (i) *pHappens(e,t,ℜ(t1,t2))*: this predicate signifies the possibility that an event e occurs at some time t within the time range ℜ (t1,t2)

(ii) *pInitiates(e,f,t):* this predicate signifies the possibility that an event e initiates fluent f at some time t within the time range $\Re$ (t1,t2)

(iii) *pTerminates(e,f,t):* this predicate signifies the possibility that an event e terminates fluent f at some time t within the time range $\Re$ (t1,t2)

(iv) *pHoldsAt(f,t):* this predicate signifies the possibility that a fluent f holds at some time t.

- $\alpha_{pHappens}$ is an axiom regarding the *p-EC* predicates that is expressed by the following formula:

$\alpha_{pHappens}$: $(\forall e{:}Event, t1,t2,t3,t4,t5,t6{:}Time)$

$Happens(e,t1,\Re(t2,t3)) \wedge t2 \leq t1 \wedge t1 \leq t3 \wedge *(\Re(t2,t3),\Re(t5,t6)) \neq \varnothing \Rightarrow$

$pHappens(e,t4, * (\Re(t2,t3), \Re(t5,t6)))$

- **(r1,r2)* is a function denoting the intersection of two time ranges that is defined as:

(i) $*(\Re(t1,t2), \Re(t3,t4)) = \Re(\max(t1,t3),\min(t2,t4))$, if $max(t1,t3) \leq min(t2,t4)$

(ii) $*(\Re(t1,t2), \Re(t3,t4)) = \varnothing$, if $max(t1,t3) > min(t2,t4)$.

To reason with the latter kind of events (i.e. events that may have occurred in a time range), all the occurrences of the EC predicates in f are replaced with the *p-EC* predicates. This re-writing creates the *pForm* of$f$. The *p-EC* predicates signify the possibility of the occurrence of an event *e* at some time point *t* within $\Re$(t1,t2) as opposed to the EC predicates that signify the definitive occurrence of an event *e* at some time point *t* within the same range with certainty. The axiom $\alpha_{pHappens}$ is then deployed to check for potential violations of f as it can be used to derive events which may have occurred in ranges which are overlapping with ranges expected by the *pForm* of$f$.

The use of the set of events $E_u(abd(f),t)$ from which entailments may be drawn in Definition 6, reflects the deployment of events that are generated by abductive reasoning in the search for potential inconsistencies. In our framework we apply a logic-based approach for abductive reasoning [Con91, Eit95, Pau93]. In standard formulation [Con91, Eit95, Pau93], abductive reasoning is described as the framework, where given a set of causes *C*, a set of effects *E* and a logical theory τ defined over some language, an explanation of a set of observations ($\Omega \subseteq$ E) is a finite set of sentences ϕ such that:

- ϕ is consistent with τ,
- $\tau \cup \phi \models \Omega$, where Ω denotes the conjunction of all $\omega \in \Omega$.

Thus, Definition 6 realises an approach where abductive reasoning is achieved through deduction using the completing formulas of the abducible formulas of a service based system

similarly to the approach applied in [Con91]. The conditions in the definition of *abd(f)* in Definition 6 ensure that the events generated by abductive reasoning are possible explanations of events that have occurred in the event log of the system. The first condition (i.e., condition (i)) is used to ensure that only explanations of events that have been generated during the operation of the system and recorded in the event log of it (as opposed to events that may be assumed by virtue of applying the principle of negation as failure or derived by some functional property or assumption) may be used in the process of detecting possible violations. The second condition (i.e., condition (ii)) is used to ensure that explanations of events will always be "minimal" (i.e., as specific as possible and the predicates that satisfy this condition are called abducible predicates [Con91]).

It should be noted that in our framework abducibles are generated without checking whether they preserve the consistency of the original theory (i.e., the behavioural properties, functional properties, QoS properties and assumptions) as in standard formulations of abductive reasoning [Con91, Eit95, Pau93]. This is because our objective is not to generate consistent explanations of system events as in pure abductive reasoning but to generate all the possible explanations of these events and then check whether these explanations violate the behavioural properties, functional properties or QoS properties of an SBS system. Finally, it should be appreciated that according to definition 6, abductive reasoning is applied only to functional properties or assumptions but not to behavioural properties. This is because, behavioural properties are extracted directly from the source code of the composition process and they represent definite sequences of events in the process execution. In other words given a behavioural property of the form $C \Rightarrow A$, it would not be necessary to generate the event C from observing event A, since in cases where the system has produced A due $C \Rightarrow A$, then C must also have been recorded in the event log of the system. In cases where A has been produced due to another formula the use of $C \Rightarrow A$ in the abductive reasoning would not be plausible.

Example. Given the event log in Figure 4.4, at $t=40$, the behavioural property *BP2* is detected to have been possibly violated. This is because the negation of the *pForm* of this statement is entailed by:

(i) the literal *pHappens(re:UI:CarRequest(op6,veh1,loc3),33,ℜ(33,33))* that can be deduced from the literal L14 in Figure 4.4

(ii) the negation of the literal *pHappens(in:IS:MakeUnAvailable(ID,veh1,,oc3),40,ℜ(40,40))* that can be deduced by applying negation as failure and from the literal L15 in Figure 4.4, and

(iii) the literals *pHappens(rc:SS:Depart(ID),t,ℜ(34,39))*, *pInitiates(rc:SS:Depart(ID), valueOf(p,loc3), t)* and *pInitiates(rc:SS:Depart(ID), valueOf(v, veh1),t)*. These literals are derived by the application of abductive and deductive reasoning as follows,

Let,

- The functional properties presented in Figure 4.3, comprise the logical theory τ.
- C = {Happens(rc:SS:Enter (_oID),t, ℜ(t,t)),
  Initiates(rc:SS:Enter(_oID), valueOf(info.v, _vID),t),
  Initiates(rc:SS:Enter(_oID), valueOf(info.p, _pID),t)}
  is a set of causes (abducibles). Entries in C come from the body of the functional property *FP4* in theory τ. This is because according to the definition 6, *FP4* $\in$ *abd(BP2)*, since *FP1* depends on the complement formula of *FP4* and *BP2* depends on *FP1*.
- E = {Happens(rc:UI:RetKey(_oID),t, ℜ(t,t)),
  Initiates(rc:UI:RetKey(_oID), valueOf(info.v, _vID),t),
  Initiates(rc:UI:RetKey(_oID), valueOf(info.p, _pID),t)}
  is a set of effects. Entries in E come from the head of the functional property *FP4* in theory τ.

If we consider L22, L23 and L24 as a set of observations , i.e. Ω =
{Happens(rc:UI:RetKey(op12),45, ℜ(45,45)), Initiates(rc:UI:RetKey(op12), valueOf(info.v, veh1),45), Initiates(rc:UI:RetKey(op12), valueOf(info.p, loc3),45)}, then
{Happens(rc:SS:Enter(ID),t, ℜ(35,44)), Initiates(rc:SS:Enter(ID), valueOf(info.v, veh1),t), Initiates(rc:SS:Enter(ID), valueOf(info.p, loc3),t)} is a set of explanations of Ω, because

- {Happens(rc:SS:Enter(ID),t, ℜ(35,44)), Initiates(rc:SS:Enter(ID), valueOf(info.v, veh1),t), Initiates(rc:SS:Enter(ID), valueOf(info.p, loc3),t)} is consistent with τ, and
- $\tau \cup$ {Happens(rc:SS:Enter(ID),t, ℜ(35,44)), Initiates(rc:SS:Enter(ID), valueOf(info.v, veh1),t), Initiates(rc:SS:Enter(ID), valueOf(info.p, loc3),t)} |= {Happens(rc:UI:RetKey(op12),45, ℜ(45,45)),
  Initiates(rc:UI:RetKey(op12), valueOf(info.v, veh1),45),
  Initiates(rc:UI:RetKey(op12), valueOf(info.p, loc3),45)}.

From

- the abduced literals Happens(rc:SS:Enter(ID),t,ℜ(35,44)),
  Initiates(rc:SS:Enter(ID), valueOf(info.v, veh1),t),
  Initiates(rc:SS:Enter(ID), valueOf(info.p, loc3),t)}

- the literals L4, L5, L6, and
- formula *FP1*

we can deduce the literals `Happens(rc:SS:Depart(ID),t, ℜ(28,43)),`
`Initiates(rc:SS:Depart(ID), valueOf(info.p,loc3), t),`
`Initiates(rc:SS:Depart(ID), valueOf(info.v, veh1),t).`

Subsequently from axiom $\alpha_{pHappens}$ we can deduce `pHappens(rc:SS:Depart(ID),t, ℜ(34,39)), pInitiates(rc:SS:Depart(ID), valueOf(info.p,loc3), t)` and `pInitiates(rc:SS:Depart(ID), valueOf(info.v, veh1),t).`

In this case, a possible inconsistency may have occurred due to a malfunctioning of the SS service that failed to generate an *rc:Depart* event within the time range *ℜ(34,39)*. The possibility of the occurrence of this event has been established by applying deductive and abductive reasoning.

4.3.5 Potentially Unjustified Behaviour

Similarly to the definition of unjustified behaviour, we define cases of potentially unjustified behaviour, as follows:

Definition 7: A behavioural property of the form *f: C⇒A* is said to generate *potentially unjustified behaviour* if and only if there is an event *e* such that:

(i) $e \in E_R(t)$

(ii) *e* can be unified with A

(iii) $\{E_R(t) - \{e\}, f, EC_a\} \models_{nf} e$

(iv) $\{E_R(t) - \{e\}, B_S - \{f\}, EC_a\} \not\models_{nf} e$ and

(v) there is a condition L in the body of the *pForm* of *f* for which, $\{E_R(t), E_U(dep(f),t), E_U(abd(f), t), AX_{pHappens}\} \models \neg$pForm(L)

According to Definition 7, a case of potentially unjustified behaviour arises when the conditions of a behavioural property which are satisfied by the recorded behaviour of a system may have been violated by the expected behaviour of it. In such cases the event generated by the head of the formula constitutes a potentially unjustified behaviour.

Example. Given the event log in Figure 4.4, at time *t=40*, the behavioural property *BP3* is satisfied by $E_R(40)$.

`Happens(re:UI:CarRequest(op6,veh1,loc3),33,ℜ(33,33)) ∧`
`¬Happens(rc:SS:Depart(ID),t,ℜ(34,39))⇒`

`Happens(in:IS:MakeAvailable(op7,veh1,loc3),40,ℜ(40,40))`

This is due to the literal L14, literal L15 and the literal ¬`Happens (rc:SS:Depart(ID), t, ,ℜ(34,39))` which can be established by the principle negation as failure. But it has to be appreciated that the absence of a `Depart` event in this instance may be the result of a malfunctioning SS service. If this is the case and an unrecorded `Depart` event has occurred then the behaviour of CRS has violated *BP3*. To cover this possibility, we apply abductive and deductive reasoning as in the example of Section 4.3.4 to derive the literal {`Happens(rc:SS:Depart(ID),t, ℜ(28,43))`}. This literal entails that the conditions of the behavioural property *BP3* are potentially unjustified at *t=40*. More specifically, the *pForm* of the second condition of *BP3* is violated (i.e. ¬`pHappens(rc:SS:Depart(ID),t,ℜ(34,39))`) by {$E_R(55)$, $E_U(dep(f),55)$, $E_U(abd(f), 55)$, $AX_{pHappens}$}. This is because we can derive `pHappens(rc:SS: Depart(ID),t,ℜ(34,39))` from `Happens(rc:SS:Depart(ID), t,ℜ(28,43))` and axiom ($\alpha_{pHappens}$).

4.4 The Monitoring Scheme

In this section we explain the monitoring scheme used to detect the property deviations in our framework. The monitoring scheme is realised by the component *monitor* of the architecture presented in Chapter 2. Figure 4.5 shows the modules deployed by the monitor for runtime monitoring along with the structural organisation of the runtime components of the

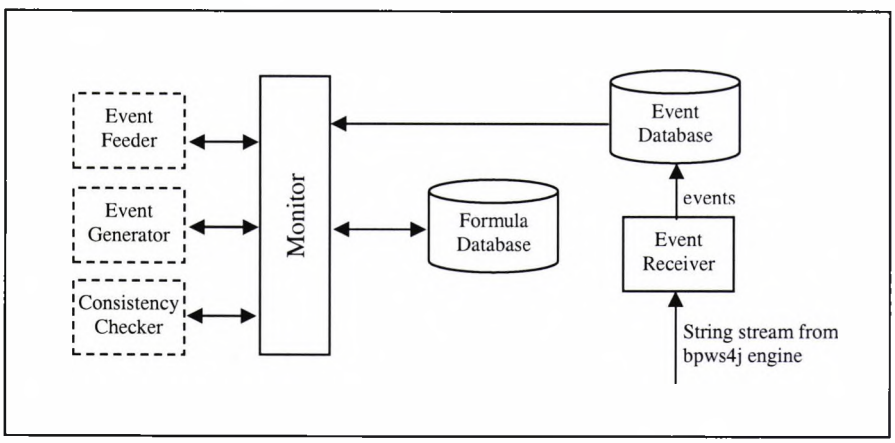


Figure 4.5: Runtime components of the monitoring architecture

architecture. The modules deployed by the monitor are shown as dotted rectangles in the figure and these modules are: (i) *Event Feeder* that handles recorded events in the monitoring

process, (ii) *Event Generator* that realises the expected behaviour of the system being monitored and (iii) *Consistency Checker* that checks the deviation. In the rest of this chapter we discuss the mechanisms of these modules that exert the functionality of the monitor.

4.4.1 The Monitor

It is discussed in Section 4.3 that the unrecorded expected behaviour of a system comprises a set of events that can be derived from a given set of formulas and these derived events are used to detect deviation of the formulas that are depended on these events. Therefore given a set of formulas[2] to be monitored, the monitor first identifies the interdependent formulas. The formula interdependency is identified according to the definition of *dep(f)* presented in Definition 4, in Section 4.3.2. The algorithm shown in Figure 4.6 is used to identify formula interdependency.

```
Algorithm: Find_InterDependent_Formulas
Input: Set of Formulas, F_S
Output: FIL is a list of labelled link.
//in this algorithm a labelled link between two formulas is denoted as follows,
// F_1 --Pr--> F_2 , Predicate Pr in formula F_2 can be derived from formula F_1
FIL:= {}
for each formula F_I: P ⇒ Q in F_S do
  for each predicate A in Q do
      for each formula F_J : R ⇒ S in F_S such that I ≠J do
        if A can be unified with a predicate B in F_J then
            add a link F_J --A--> F_I in FIL
        end if
      end for
  end for
end for
```

Figure 4.6: Formula interdependency identification algorithm

For example, if the set of monitorable formulas F_S comprises the behavioural properties in Figure 4.2 and the functional properties in Figure 4.3, the above algorithm would generate a list of labelled links which are shown as a labelled directed graph in Figure 4.7.

At runtime, the monitor maintains templates that represent different instantiations of the formulas that specify the behavioural properties, functional properties, QoS properties and assumptions for a system. An instantiation of a formula represents a copy of the formula having the variables (or a subset of the variables) bound to specific values.

[2] The monitoring scheme assumes that the predicates in the body and head of a formula are combined by conjunctive logical operator (∧) only and non time variables are universally quantified.

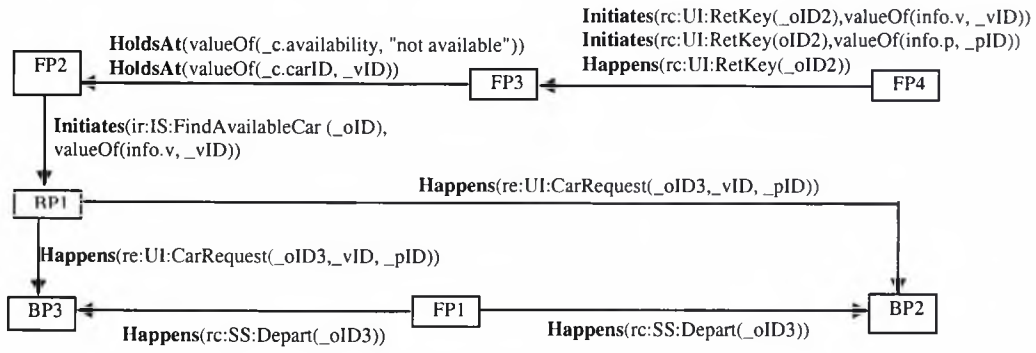


Figure 4.7: Formula interdependency graph

The templates maintained by the monitor store the state of different instantiations of a formula *f*, including:

- The identifier (*FID*) and type of *f*. The type of *f* is *F* (future) if all the predicates *p* in *f* whose time variables are constrained by unconstrained time variables of other predicates must occur after these predicates (e.g., *BP2*). The type of *f* is *P* (past) if there is at least one predicate *p* that must occur before another predicate *q* that constraints it (e.g., formulas *FP1*).
- A dependants list (*DP*) of *(id, P)* which indicate other formulas which depend on *f* (*id* is the identifier of a formula that depends on *f* and *p* is the predicate that creates the dependency (see Definition 4)),
- The bindings (*VB*) of the non-time variables of all the predicates in *f*, and
- For each predicate *p* in *f*:
  - The *quantifier* of the time variable (Q) and signature (SG) of *p*.
  - A *time range* (*LB*, *UB*) that indicates when *p* should occur. The boundaries of this range are set according to the time constraint of *p* in *f*.
  - The *truth-value (TV)* of *p* which can be: *UK* (if the truth-value of p has not been established), *True* (if *p* is true), or *False* (if *p* is false).
  - A *time stamp* (*TS*) that indicates the time at which the truth-value of *p* is established.
  - The *source* (SC) of the evidence for the truth value of *p* which can be: *UK* (if the truth value of *p* has not been established), *RE* (if the truth value of *p* is established by a recorded event unified with it), *DE* (if the truth value of *p* is established by a derived event unified with it), or *NF* (if the truth value of *p* is established by the principle of negation as failure)

For example an initial template for the formula BP1 would look as follows,

ID	BP1						
T	F						
DP	(BP2, Happens(re:UI:CarRequest(_oID,_vID, *pID*)) , *(BP3, Happens(re:UI:CarRequest(*_oID,_vID,_pID))						
VB							
P	Q	SG	TS	LB	UB	TV	SC
1	∀	**Happens**(rc:UI:CarRequest(_oID1),t1,ℜ(t1,t1))	t1	t1	t1	UN	UN
2	∃	**Initiates**(rc:UI:CarRequest(_oID1), valueOf(info.p,_pID),t1)	t1	t1	t1	UN	UN
3	∃	**Happens**(in:IS:FindAvailable(_oID2,_pID),t2, ℜ(t1,t2))	t2	t1	t2	UN	UN
4	∃	**Happens**(ir:IS:FindAvailable(_oID2),t3, ℜ(t2,t3))	t3	t2	t3	UN	UN
5	∃	**Initiates**(ir:IS:FindAvailable(_oID2),valueOf(info.v,_vID),t3)	t3	t3	t3	UN	UN
6	∃	**Happens**(re:UI: CarRequest(_oID3,_vID,_pID),t4, ℜ(t3,t3 + t_u))	t4	t3	t3 + t_u	UN	UN

The algorithms used by the monitor, the event feeder, the event generator and the consistency checker to realise the monitoring process are presented in the rest of this chapter in some pseudo code language. A textual description of each algorithm is followed by each algorithm. Before presenting the algorithms in the subsequent sections we define the basic data structures and constants used in the algorithms.

4.4.1.1 Basic Definitions

Following constants, data structures and definitions are used in all the algorithms presented in the sequel

Constants:

SAT: Satisfied

I_R_B: Inconsistency with Recorded Behaviour

I_E_B: Inconsistency with Expected Behaviour

U_B: Unjustified Behaviour

P_I_E_B: Possible Inconsistency with Expected Behaviour

P_U_B: Potentially Unjustified Behaviour

RE: Recorded Event

DE: Derived Event

NF: Negation as Failure

UK: Unknown

MON_REC: monitoring with respect to recorded events only

MON_MIX: monitoring with respect to recorded and derived events.

Min_t: the minimum time between the occurrences of two events

Data Structures:

Link structure has the following fields:

- *Id:* holds a formula ID
- *P:* holds a predicate signature

Variable structure has the following fields:

- *N:* holds name of a variable
- *T:* holds the type of the variable

Time Variable structure has the following fields:

- *N:* holds name of a time variable
- *V:* holds the value of the time variable

Time Expression structure

This structure holds an arithmetic expression, that consists of time variables, e.g. T1, T2 etc., arithmetic operators e.g. +, -, *, / and numbers. It has the following fields,

- T_i: a list of *Time Variable* structure that holds the time variables.
- OP_i: a list of operators
- N_i: a list of numbers.

Template structure has the following fields:

- *Fid:* Formula ID, holds the ID of the formula
- *T:* Type, holds the type of the formula, could be 'F 'for future and 'P' for past
- *DP:* Dependants, a list of Link structure
- *body:* Predicates, a collection of Predicate structure
- *head:* Predicates, a collection of Predicate structure
- P_n: a collection of references to Predicate structures, entries in P_n refer to all the predicates in *body* and *head.*
- *updated:* a Boolean value, set to true if the template has been updated, else set to false
- *forChecking*: a Boolean value, set to *true* if the template should be considered for consistency checking (e.g. in case of behavioural properties, functional properties, and QoS properties), else set to *false* (in case of assumptions).
- *forDeduction*: a Boolean value, set to *true* if the template should be used as deductive rule to generate derived events.
- *ST:* Status, Holds the monitoring status (e.g. I_R_B, I_E_B etc.) of the template, Initially *UK.*
- u_f: Variable Bindings. See below for the definition of u_f.

Predicate structure has the following fields:

- *SG:* Signature, holds the predicate signature
- *VL:* Variable of this predicate. Holds a list of *Variable* structure.
- *Q:* Quantifier, holds quantifier (existential/universal) information
- *NoQ:* Negation on Quantifier, holds true if there is negation on quantifier, else holds false
- *TV:* Truth Value, holds truth value of the predicate.
- *LB:* Lower Bound, a *Time Expression* structure that indicates the earliest time that predicate should occur
- *UB:* Upper Bound, a *Time Expression* structure that indicates the latest time that the predicate should occur
- *TS:* Time Stamp, a *Time Variable* structure that keeps predicate occurrence time, initially value is set to $LB\text{-}min_t$
- *SC:* Source, holds source of event
- *pForm:* initially holds false, set to true if pForm condition occurs

Event structure has the following fields:

- *SG:* Signature, holds the event signature
- *NG:* Negated, holds true if the event denotes a negated predicate, else set to false
- *VB:* Variable Bindings, a collection of *(v, c)* pair, where *v* is a variable structure and *c* is the value assigned to it.
- *TS:* Time Stamp, occurrence time of the event

Definitions

u_f: u_f is the current most general unifier of the variables $\{v_1, \ldots, v_n\}$ of an instance of a formula *f* represented by a template *t* that is defined as $\{(v_1, c_1), \ldots, (v_m, c_m)\}$ where v_i is a variable of *f* and c_i is a constant value assigned to it. In general, we have that $m \leq n$ as there may be variables of *f* which have not been assigned to any values at a specific instance of time.

$u_{f|P}$: $u_{f|P}$ is the projection of u_f over the variables of a predicate *P* of *f* defined as: $u_{f|P} := \{(v_i,c_i) \mid (v_i \in P.VL) \text{ and } ((v_i,c_i) \in u_f)\}$

partial($u_{f|P}$): a boolean function that returns **True** if there is variable v in P.VL for which there is no pair (v,c) in $u_{f|P}$ and **False** otherwise

imgu(e,P,$u_{f|P}$):a function that returns the most general partial unifier (i.e., a set of the form $\{(v_1, c_1), \ldots, (v_k, c_k)\}$) of a predicate *P* with an event *e* that is compatible with the current unification of the variables of P, $u_{f|P}$, or an empty set if no such unifier exists.

eval(*TimeExpression*) : a function that evaluate a time expression and returns the value.

4.4.1.2 The Monitoring Algorithm

The monitor creates two templates for each formula to check for different types of deviations from the formula and to derive events from it. The first of these templates is used to detect deviations with respect to recorded events only and is filled only with recorded events. The second template is used to detect deviation with respect to recorded and derived events and is filled with both recorded and derived events.

The monitor picks an event from the event database and iterates over each template to process the event[3]. For each template that it visits, the monitor invokes the *event feeder* by calling the procedure *feed*, to feed the event to the template. The *event feeder* is responsible for updating the truth-value of each predicate in the template that can be unified with the event and to create new instances of the template if required. Section 4.4.2 describes the algorithm for *event feeder*. If a template instance is updated during the execution of the *feed* procedure, the monitor notifies the *event generator* and the *consistency checker* about the update of the template. Once notified about the update of a template, the *Event Generator* is responsible for generating any derived event that can be deduced from the template. Section 4.4.3 describes the algorithm for *event generator*. The *Consistency Checker* is responsible to check if there is a violation of the formula represented in the template. Section 4.4.4 describes the algorithm for *consistency checker*.

The monitoring algorithm is shown in Figure 4.8. In line 1 an empty set R_{temps} is declared that holds the templates used only for recorded events. In line 2 an empty set M_{temps} is declared that holds the templates used for both recorded and derived events. In line 3 the monitoring mode is initialised. The *for* loop in line 4, creates an empty template for each formula to be monitored and the template is added to R_{temps} and M_{temps} respectively in lines 6 and 7. In line 9 and 10 the templates are stored in the database.

[3] The monitoring scheme assumes that the events are stored in the database in order of their occurrence.

```
Monitoring()
1. R_temps := {}
   /* R_temps is a template set representing instantiations of formulas using
   only recorded events */
2. M_temps := {}
   /* M_temps is a template set representing instantiations of formulas using
    recorded and/or derived events */
3. initialize mode
    /* mode signifies if the monitoring will be wrt recorded event only or
       wrt recorded and derived events */
4. for each formula f to be monitored do
5.       create an empty template t for f
6.       add t to R_temps
7.       add t to M_temps
8. end for
9.   store templates in R_temps in database
10. store templates in M_temps in database
11. events:= {} /* holds events to be processed next */
12. lastEventTime:= 0 /* holds the time stamp of the latest event being
                         processed */

13. while (there are more events to handle) do
14.     receive the next event e from Event Database
15.     if e.TS > lastEventTime then
16.           events:={}
17.     end if
18.     lastEventTime := e.TS
19.     add e at the beginning of events
20.     Templates := {} /* templates to be considered for the event */
21.     T_d:={} /* templates to be removed after processing this event */
22.     T_a:={} /* new templates created after processing this event */
23.     for each event e_i in events do
24.         if mode=MON_REC then
25.              Templates:=R_temps /* retrieve templates from database */
26.         else
27.              Templates:=M_temps /* retrieve templates from database */
28.         end if
29.         for each template t in Templates do
30.                feed(t, e_i, t_add, t_del)
31.                if t.updated = true then
32.                    t.updated := false
33.                    check_consistency(t, Templates)
34.                    if mode = MON_MIX and t.forDedcution = true then
35.                       generate_event(t, Templates)
36.                    end if
37.                end if
38.                T_a := T_a ∪ t_add
39.                if mode = MON_REC then T_d := T_d ∪ t_del end if
40.         end for
41.         insert all templates of T_a into Templates
42.         delete all templates of T_d from Templates
43.         store templates in Templates in database
44.     end for /* end of each event e_i */
45. end while
End Monitoring
```

Figure 4.8: The monitoring algorithm

The *while* loop starts in line 13 drives the whole monitoring process and it iterates for each single event to be considered in the database. In line 14, an event *e* is received from the database. The codes between lines 15 and 19, accumulates the events that have same time stamp. This accumulation is necessary from the consideration that our monitoring scheme is

incremental and it assumes that the events are serialised in the database according to the time of occurrence. When an event is processed by the monitor it assumes that the impact of the occurrence time of all the previous events on the formulas has been considered. But this can not be guaranteed in case of events with the same time stamp which may require reprocessing of the events with the same time stamp. Between lines 24 and 28 templates are retrieved from the database depending on the monitoring mode. For each event *e* the *for* loop in line 29 iterates over each template instance *t*. In line 30, the *feed* method in the *event feeder* is called to feed the event *e* to the template *t*. While an event is being processed by the *feed* method, the method may also generate new template instances or identify a list of templates to be removed. The *event feeder* marks a template as removable if a monitoring decision from the template instance can not be made any more. The codes between lines 31 and 36 invoke the *event generator* and the *consistency checker* if the *event feeder* has updated the template *t*. In line 41 all the new template instances created by the *event feeder* are added to the template set. In line 42 all the templates marked as removable by the *event feeder* are removed from the template set. In line 43 the template set is saved in the database.

4.4.2 The Event Feeder

The *event feeder* receives an event and a template instance from the monitor and checks if the template instance should be updated by the event. Updates may be made if the signature, the event variable bindings and the time of the event comply with the predicate signature, the predicate variable bindings, and the time range of the predicate in the template instance, respectively. If a predicate is updated, the bindings of the predicate's variables in the template instance are also updated. New instances of templates may also be generated if the event corresponds to an unconstrained predicate of the template instance (i.e., a predicate whose time variable is not constrained by the time variable of another predicate), or the variable bindings of the predicate have values that are different from the event variable bindings values. The truth-value of a predicate in the template instance may also be updated by applying the principle negation as failure.

4.4.2.1 Overview of the Event Feeding Algorithm

The *event feeder* receives an event and a template and updates the truth-value of the predicate in the template if the predicate can be unified with the event and its truth-value has not been set yet. An event can be unified with a predicate if the signature, the event variable bindings comply with the predicate signature, the predicate variable bindings. Once the unification is done the *event feeder* checks if the event time stamp complies with the predicate time range.

This checking considers the quantifier on the predicate's time variable and the time range of this variable. Depending on the quantifier of their time variables, predicates are distinguished into two types: (i) predicates with existentially quantified time variables and (ii) predicates with universally quantified time variables.

Table 4.1 shows all the possible cases of updating existentially and universally quantified predicates by events which are considered by the event feeder. In the table, an EC predicate (e.g. **Happens**, **Initiates** etc) is signified by $P(x, t_p, [LB,UB])$, where P stands for the predicate signature, x represents the variable (if any) in the predicate, t_p is the time stamp of the predicate and $[LB,UB]$ is the time range in which P should occur. Predicates are shown under the predicate column. An event is shown as $P(a, t_e)$ or $Q(a, t_e)$, where P or Q stands for the event signature, a represents the value of the variable in the event and t_e is the occurrence time of the event. Events are shown under the event column. The second row under the event column shows the checking of the event time stamp compliance with the predicate time range and other conditions that need to be checked to update the truth-value of a predicate. Each cell numbered between 1 and 24 signifies a possible case to be considered and shows some of the updates to be made in the predicate.

The update procedure depends on the type of the quantification of a predicate as it is explained below

Existentially quantified predicates:

(i) A predicate of the form $(\exists t_p).P(\underline{x},t_p,[LB,UB])$ signifies that there is at least one single time point t_p in the range $[LB,UB]$ at which an event that can be unified with P has occurred. This truth-value is set to *true* as soon as the first event that can be unified with P occurs between LB and UB (see cases 1 and 7 in Table 4.1). The occurrence of an event between LB and UB that can not be unified with P (or can be unified, but the event is negated) does not have any impact on the truth value of the predicate (see cases 2 and 8 in Table 4.1). The absence of an event unifiable with P within the time range from LB to UB is confirmed as soon as the first event that cannot be unified with P occurs after UB and the truth-value of P is set to *false* at the time of the occurrence of this event if the truth-value of P is still undefined (see cases 3 and 9 in Table 4.1).

(ii) A predicate of the form $\neg(\exists t_p).P(\underline{x},t_p,[LB,UB])$ signifies there does not exist any single time point t_p in the range $[LB,UB]$ such that the evidence of predicate P can be

found. The truth-value of a predicate of this form is set to *false* as soon as the first event that can be unified with *P* occurs between *LB* and *UB* (cases 4 and 10 in Table 4.1). The occurrence of an event between *LB* and *UB* that can not be unified (or can be unified, but the event is negated) with *P*, does not have any impact on the truth value of the predicate (cases 5 and 11 in Table 4.1). If no event unifiable with *P* occurs between *LB* and *UB*, the absence of an event unifiable with *P* is confirmed as soon as the first event that cannot be unified with *P* occurs after UB and the truth-value of the predicate is set to *true* if the truth-value of *P* is still undefined (cases 6 and 12 in Table 4.1).

Table 4.1: Possible predicate updating cases considered by the Event Feeder

Predicate	**Event**		
	$P(a, t_e)$	$\neg P(a, t_e)$ / $Q(a, t_e)$	
	Condition: LB <= t_e <= UB && P.TV=UK	Condition: LB <= t_e <= UB && P.TV=UK	Condition: UB < t_e && P.TV=UK
$\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t	1 P.TV:=true x:=a t_p:= t_e	2 No effect on P.TV	3 P.TV:=false t_p:= t_e
$\neg\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t $\forall t. \neg P(\underline{x}, t_p, [LB,UB])$	4 P.TV:=false x:=a t_p:= t_e	5 No effect on P.TV	6 P.TV:=true t_p:= t_e
$\exists t_p.P(\underline{x}, t_p,[LB,UB])$, LB=$t_1$ UB=t_2	7 P.TV:=true x:=a t_p:= t_e	8 No effect on P.TV	9 P.TV:=false t_p:= t_e
$\neg\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2 $\forall t. \neg P(\underline{x}, t_p, [LB,UB])$	10 P.TV:=false x:=a t_p:= t_e	11 No effect on P.TV	12 P.TV:=true t_p:= t_e
$\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t	13 P.TV:=true x:=a t_p:= t_e	14 No effect on P.TV t_p:= t_e	15 P.TV:=false t_p:= t_e
$\neg\forall t_p.P(\underline{x}, t_p,[LB,UB])$, LB=t UB=t $\exists t. \neg P(\underline{x}, t_p, [LB,UB])$	16 No effect on P.TV t_p:= t_e	17 P.TV:=true x:=a t_p:= t_e	18 P.TV:=false t_p:= t_e
$\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2	19 P.TV:=true x:=a t_p:= t_e	20 No effect on P.TV t_p:= t_e	21 P.TV:=false t_p:= t_e
$\neg\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2 $\exists t. \neg P(\underline{x}, t_p, [LB,UB])$	22 No effect on P.TV t_p:= t_e	23 P.TV:=true x:=a t_p:= t_e	24 P.TV:=false t_p:= t_e

Universally quantified predicates:

(i) A predicate of the form $(\forall t_p).P(\underline{x},t_p,[LB,UB])$ signifies at every observable time point t_p in the range $[LB,UB]$ predicate P is true. The truth-value of a predicate of this form is set to *true* if any event that can be unified with P occurs between LB and UB (cases 13 and 19 in Table 4.1). The occurrence of an event between LB and UB that can not be unified (or can be unified, but the event is negated) with P, does not have any impact on the truth value of the predicate (cases 14 and 20 in Table 4.1), but the time stamp of P should be updated upon the occurrence of such an event in order to indicate the latest time point in the range $[LB,UB]$ at which the predicate has not been satisfied yet. If no event unifiable with P occurs between LB and UB, the absence of an event unifiable with P is confirmed as soon as the first event that cannot be unified with P occurs after UB and the truth-value of the predicate is set to *false* if the truth-value of P is still undefined (cases 15 and 21 in Table 4.1).

(ii) A predicate of the form $\neg(\forall t_p).P(\underline{x},t_p,[LB,UB])$ signifies that not for all observable time points throughout the range $[LB,UB]$ P is true. The truth-value of a predicate of this form is set to *true* as soon as the first time point within (LB, UB) at which there is no event that can be unified with P occurs (see cases 17 and 23 in Table 4.1). The occurrence of an event between LB and UB that can be unified with P, does not have any impact on the truth-value of the predicate (cases 16 and 22 in Table 4.1), but the time stamp of P should be updated to keep track of all time points in the range $[LB,UB]$. However, as soon as the first event that cannot be unified with P occurs after UB the truth-value of the predicate is set to *false* if the truth-value of P is still undefined (see cases 18 and 24 in Table 4.1).

When the *event feeder* updates the truth-value of a predicate it also updates the bindings of the variables of other predicates in the template. For example, consider the following formula,

(4.1) $(\forall$ x, t1)Happens(ic:S1:P(ID, x),t1,$\Re$(t1,t1)) $\Rightarrow$ ($\exists$ t2)Happens(ic:S1:Q(ID, x, y),t2, $\Re(t1+t_u,t1+5*t_u)$)

If the *event feeder* receives the event *ic:S1:P(id1,a)* at time t=3, it will set the truth-value of the predicate *Happens(ic:S1:P(ID, x),t1,$\Re$(t1,t1))* to *true* at the same time the variable x in the predicate *Happens(ic:S1:Q(ID, x, y),t2, $\Re(t1+t_u,t1+5*t_u)$)* will be bound to value a.

When the *event feeder* updates the truth-value of a predicate P it also updates the upper limit and lower limit of the ranges of any other predicates in the template that depend on the time variable of the predicate P that has been updated. Consider, for example, the formula 4.1, if

the *event feeder* receives the event *ic:S1:P(id1,a)* at time 3, it will set the range of the predicate *Happens(ic:S1:P(ID, x),t1,ℜ(t1,t1))* to *ℜ(3,3)* and the range of the predicate *Happens(ic:S1:Q(ID, x, y),t2, ℜ(t1+t_u,t1+5*t_u))* to *ℜ(4,8)* (assuming t_u=*1*). This is because the time range of the latter predicate depends on the time variable of *Happens(ic:S1:P(ID, x),t1,ℜ(t1,t1))*.

The *event feeder* creates new instances of a template if an event that can be unified with an unconstrained predicate of it occurs (i.e., a predicate whose time variable is not constrained by the time variable of another predicate), or the variable bindings of the predicate have values that are different from the event variable bindings values. In the case of formula 4.1, for example, if the *event feeder* receives the event *ic:S1:P(id1,a)* at time 3, the *event feeder* will create a new instance of template (since *P* is an unconstrained predicate), set the truth-value of this predicate to *true*, and update the variable bindings and time ranges of P and other predicates.

Following the update of formula 4.1 due to the event *ic:S1:P(id1,a)* at time 3, the event feeder would create the following template (instance I1) to represent the instance of the formula that is created due to the event,

ID	BP1						
T	F						
DP							
VB	(x, a)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:S1:P(ID, x),t1,ℜ(t1,t1))	3	3	3	True	RE
2	∃	**Initiates**(ic:S1:Q(ID, x, y), t2, ,ℜ(t1+ t_u,t1+5* t_u))	t2	4	8	UK	UK

Then if at time t=5, the *event feeder* receives the event *Happens(ic:S1:Q(id2,a,b))* it would create another template instance *I2* from *I1* and update the truth value of the predicate *Happens(ic:S1:Q(ID, x, y), t2,ℜ(t1+t_u,t1+5*t_u))* in *I2*. The template instance *I2* would look as follows,

ID	BP1						
T	F						
DP							
VB	(x, a), (y, b)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:S1:P(ID, x),t1,ℜ(t1,t1))	3	3	3	True	RE
2	∃	**Initiates**(ic:S1:Q(ID, x, y), t2, ,ℜ(t1+ t_u,t1+5* t_u))	5	4	8	True	RE

Note that in this, case the *event feeder* does not update directly the template instance *I1*. Instead it creates a new instance *I2* as a copy of I1 and updates *I2*. This is because *I1* should be used to create another template instance if the feeder receives another event *Q* and in this event the variable *y* had a value other than *b*, i.e. an event like *Happens(ic:S1:Q(id2,a,c))*. In

this case, a new template instance could not be created from *I2* as all the variables in it are bound to some value. Thus, any new instance of formula 4.1 should be created from *I1* as the variable *y* in *I1* is not bound to some value.

4.4.2.2 The Algorithm

Figure 4.9 shows the algorithm for the *event feeder*,

```
feed(T, E, T_add, T_del)
/* E: event, T: formula template */
/* T_add is a list of new templates created from T after processing event E */
/* T_del is a list of templates to be pruned after processing event E */
1. T_add:={}, T_del:={}
2. for each predicate P in T such that (P.TV=UK) do
3.    u_cur := imgu(E,P,u)
4.    if(P.tv is unconstrained) then
5.      if(u_cur ≠ ∅) then
6.       if u_cur - u_f|P ≠ ∅ then
          /*create a copy of T if an additional variable of P is unified with
               the current event */
7.          T' := T
8.          T_add := T_add ∪ {T'}
9.       end if
10.      P.TV:= ¬P.NoQ
11.      P.SC := RE
12.      P.TS.V := E.TS
13.      T.U_f := T.U_f ∪ u_cur
14.      for each predicate P1 in T do
15.        if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N then
16.                  P1.UB.T_i.V:=P.TS.V
17.        end if
18.        if P1.LB.T_i is a time var in P1.UB such as P1.LB.T_i.N=P.TS.N then
19.                  P1.LB.T_i.V:=P.TS.V
20.        end if
21.      end for
22.      T.updated:=true
23.    end if
24. else /* constrained predicate */
25.  if(P.Q=existential and T.U_f ≠ ∅) then
      /* ∃t.p(x,t) or ¬∃t.p(x,t) predicate */
26.    if eval(P.LB) ≤ E.TS ≤ eval(P.UB) then /* E inside P's range */
27.      if u_cur≠∅ then
28.        if u_cur - u_f|P ≠ ∅ then
            /*create a copy of T if an additional variable of P is unified */
29.          T' := T;
30.          T_add := T_add ∪ {T'}
31.        end if
32.        P.TV := ¬ P.NoQ
33.        P.SC := RE
34.        P.TS.V := E.TS
35.        T.U_f := T.U_f ∪ u_cur; T.updated:=true
36.        for each predicate P1 in T do
37.         if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N then
38.                  P1.UB.T_i.V:=P.TS.V
39.         end if
40.         if P1.LB.T_i is a time var in P1.UB such as P1.LB.T_i.N=P.TS.N then
41.                  P1.LB.T_i.V:=P.TS.V
42.         end if
43.        end for
44.      end if
```

```
45.    else if E.TS > eval(P.UB)then /* E outside P's range*/
46.      if not  partial(u_f|P)  then
47.        P.TV := P.NoQ
48.        P.SC := NF;
49.        P.TS.V := eval(P.UB)
50.        for each predicate P1 in T do
51.         if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N then
52.                P1.UB.T_i.V:=P.TS.V
53.         end if
54.         if P1.LB.T_i is a time var in P1.LB such as P1.LB.T_i.N=P.TS.N then
55.                P1.LB.T_i.V:=P.TS.V
56.         end if
57.        end for
58.        T.updated:=true
59.      else /* predicate is still partial */
60.        T_del := T_del ∪ {T}
61.      end if  /*test for not  partial(u_|P) */
62.     end if   /* test for E's time range */
63.  else /* predicates: ∀t.p(x,t) OR ¬ ∀t.p(x,t)(= ∃t. ¬p(x,t)) */
64.    if eval(P.LB) ≤ E.TS ≤ eval(P.UB) then
65.      if u_cur ≠ ∅ then /* E can be unified with the current predicate */
66.        if P.NoQ=false then/* predicates: ∀t.p(x,t) */
67.           if E.NG=flase then
68.            if u_cur − u_f|P ≠ ∅ then
                 /*create a copy of T if an additional variable of P is
                                  unified */
69.              T' := T;
70.              T_add := T_add ∪ {T'}
71.            end if
72.            P.TV := ¬P.NoQ
73.            P.SC := RE
74.            P.TS.V := E.TS
75.            T.U_f := T.U_f ∪ u_cur
76.            for each predicate P1 in T do
77.             if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N
                                                                        then
78.                    P1.UB.T_i.V:=P.TS.V
79.             end if
80.             if P1.LB.T_i is a time var in P1.LB such as P1.LB.T_i.N=P.TS.N
                                                                        then
81.                     P1.LB.T_i.V:=P.TS.V
82.             end if
83.            end for
84.           else /* E.NG = true */
85.             P.TS.V:=E.TS
86.           end if /* E.NG check */
87.           T.updated := true
88.         else /* ¬ ∀t.p(x,t) predicates  */
89.           if E.NG=false then /* negated event */
90.               P.TS.V := E.TS
91.           else /* E.NG=true */
92.                 if u_cur − u_f|P ≠ ∅ then
                    /*create a copy of T if an additional variable of P is
                                                        unified */
93.                   T' := T;
94.                   T_add := T_add ∪ {T'}
95.                 end if
96.                 T.U_f := T.U_f ∪ u_cur
97.                 P.TV := P.NoQ
98.                 P.SC := NF;
99.                 P.TS.V := eval(P.UB)
100.                for each predicate P1 in T do
101.                  if P1.UB.T_i is a time var in P1.UB such as
                                                  P1.UB.T_i.N=P.TS.N then
102.                       P1.UB.T_i.V:=P.TS.V
103.                  end if
```

```
104.                    if P1.LB.T_i is a time var in P1.LB such as
                                                  P1.LB.T_i.N=P.TS.N then
105.                          P1.LB.T_i.V:=P.TS.V
106.                    end if
107.                  end for
108.                  T.updated:=true
109.              end if /* E.NG check */
110.           end if /* P.NoQ check */
111.       else /* q(_,t) events  within P's time range */
112.         if P.NoQ=false then /*  ∀t.p(x,t) predicates */
113.             P.TS.V:=E.TS
114.         else /* P.NoQ=true ¬ ∀t.p(x,t) predicates */
115.           if not  partial(u_f|P)  then
116.             P.TV := P.NoQ
117.             P.SC := RE
118.             P.TS.V := E.TS
119.             for each predicate P1 in T do
120.              if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N
                                                                        then
121.                   P1.UB.T_i.V:=P.TS.V
122.              end if
123.              if P1.LB.T_i is a time var in P1.LB such as P1.LB.T_i.N=P.TS.N
                                                                        then
124.                   P1.LB.T_i.V:=P.TS.V
125.              end if
126.             end for
127.             T.updated := true
128.           else  /* predicate still partial */
129.               T_del := T_del ∪ {T}
130.           end if /* is partial */
131.    else if E.TS > eval(P.UB)then /* event outside P's time range */
132.      if not  partial(u_f|P)  then
133.        P.TV:=false
134.        if P.TS.V + min_t=E.TS then
135.          P.SC := RE
136.        else
137.          P.SC := NF
138.        end if
139.        P.TS.V:=E.TS
140.        for each predicate P1 in T do
141.            if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N
                                                                        then
142.                P1.UB.T_i.V:=P.TS.V
143.            end if
144.            if P1.UB.T_i is a time var in P1.UB such as P1.UB.T_i.N=P.TS.N
                                                                        then
145.                     P1.LB.T_i.V:=P.TS.V
146.            end if
147.        end for
148.        T.updated := true
149.       else  /* predicate still partial */
150.            T_del := T_del ∪ {T}
151.       end if /* is partial */
152.     end if /* eval(P.LB) <= E.TS <= eval(P.UB) */
153.    end if /* universal */
154.   end if /* constrained predicate  */
155. end for
end feed
```

Figure 4.9: Algorithm for the event feeder

4.4.2.3 Explanation of the Algorithm

The event feeder receives a template T, an event E and two lists of templates T_{add} and T_{del}. T_{add} accumulates the templates created from T as result of processing the event E and T_{del} accumulates the templates that should be removed after processing the event E.

In line 1 T_{add} and T_{del} are initialised to empty list. The *for* loop starts in line 2 iterates over each predicate P in the template T and the body of the *for* loop updates the truth-value of P. In line 3 the current most general unifier u_{cur} for P and the event E is computed. If E can not be unified with P, then u_{cur} would be empty.

To update the truth-value of the predicates the algorithm first considers the unconstrained predicates between lines 4 and 23. Constrained predicates are considered between lines 24 and 154. More specifically existentially quantified predicates are considered between lines 25 and 62, universally quantified predicates are considered between lines 63 and 154.

In case of constrained predicates, the *if* condition in 5 checks if P can be unified with E. The body of the *if* condition updates the truth-value of P. In line 6 check is made to see if a new instance of the template needs to be created. This checking uses $u_{f|p}$, which holds the variable bindings that P has at the latest. This checking verifies if an additional variable in P can be bound to some value (this is done by taking the set difference of u_{cur} and $u_{f|p}$). If these condition is true, a new template instance is created in line 7 and added to T_{add} in line 8. Truth-value, source, time stamp and bindings for variables of P are updated in lines 10, 11, 12 and 13 respectively. The *for* loop between lines 14 and 21 updates the time range of other predicates in the template that depends on the time variable of P.

The code segment between lines 25 and 62 considers that P has existentially quantified time variable, i.e. cases from 1 to 12 presented in Table 4.1 are considered in this code segment. The *if* block between lines 26 and 44 considers the case where E can be unified with P, and the time stamp of E complies with the time range of P (cases 1, 4, 7 and 10 in Table 4.1). A new template instance is created (if it is needed) and added to T_{add} between lines 28 and 31. Truth-value, source, time stamp and bindings for variables of P are updated in lines 32, 33, 34 and 35 respectively. The *for* loop between lines 36 and 42 updates the time range of other predicates in the template that depends on the time variable of P. The *else* block between lines 45 and 62 considers the cases 3, 6, 9 and 12 in Table 4.1 (i.e. time stamp of E is greater than the upper limit of the range of P). In such cases, if P is not partial (i.e. all variables of P are bound to some values) then the truth-value, source, time stamp of P should be updated. This

is performed between lines 47 and 49. If P is partial then it is not possible to make any decision from this template (since it is guaranteed that the truth-value of P can not be updated by recorded events any more) and it should be removed, hence T is added to T_{del} in line 60. It should be noted that the cases 2, 5, 8 and 11 do not have any impact on the truth-value and time stamp of a predicate, therefore these cases have not been considered in the algorithm.

The code segment between lines 63 and 154 considers that P has universally quantified time variable, i.e. cases from 13 to 24 presented in Table 4.1 are considered in this code segment. The *if* block between lines 64 and 130 treats the case where the time stamp of E is within the time range of P. The *if* block between lines 66 and 88 considers the case where E can be unified with P, E is not negated and P is not negated (i.e. cases 13 and 19 in Table 4.1). A new template instance is created (if it is needed) and added to T_{add} between lines 68 and 71. Truth-value, source, time stamp and bindings for variables of P are updated in lines 72, 73, 74 and 75 respectively. The *for* loop between lines 76 and 83 updates the time range of other predicates in the template that depends on the time variable of P. The code segment between lines 89 and 91 considers the case where E can be unified with P, E is not negated and P is negated (i.e. cases 16 and 22 in Table 4.1). In this block the time stamp of P is incremented to the next time point. The code segment between liens 91 and 109 considers the case where E can be unified with P but E is negated and P is negated (i.e. cases 17 and 23 in Table 4.1). A new template instance is created (if it is needed) and added to T_{add} between lines 92 and 95. Truth-value, source, time stamp and bindings for variables of P are updated in lines 96, 97, 98 and 99 respectively. The *for* loop between lines 100 and 107 updates the time range of other predicates in the template that depends on the time variable of P. The else block between lines 111 and 130 considers the cases where E can not be unified with T (cases 14, 17, 20, 23). In such cases if P is not negated then the time stamp of P should be updated, which is done in line 113. And if P is negated and not partial (i.e. all variables of P are bound to some values) then the truth-value, source, time stamp of P should be updated. This is performed between lines 115 and 126. If P is partial then it is not possible to make any decision from this template and it should be removed, hence T is added to T_{del} in line 129.

The code segment between lines 131 and 151 considers the cases where the time stamp of E is greater than the upper limit of the range of P (i.e. cases 15, 18, 21 and 24 in Table 4.1) and P is not partial (i.e. all variables of P are bound to some values). In such cases the truth-value, source, time stamp of P should be updated. This is performed between lines 133 and 139.

4.4.3 The Event Generator

The *event generator* receives a template *T* and generates an event for each predicate in the head of the template *T* if the conditions for generating such events are satisfied. It also updates the truth-value of the predicates that correspond to the generated events in the templates that are dependent on the template *T*.

4.4.3.1 Overview of the Event Generation Algorithm

For a given template, the *event generator* generates a monitoring event if the following conditions hold (see definition 2, $\{E_R(T),f\}|=_{nf} e\}$):

(i) The predicate is in the head of a template.

(ii) The truth value of the predicate is *unknown* (UK), and all the variables of the predicate have been bound to concrete values and some other template depends on this predicate.

(iii) The truth value of each other predicate in the body of the template is *True* and only recorded events are used to update the truth values of these predicates.

It should be appreciated that the timestamp of a predicate generated by the *event generator* might not be a fixed time point in all cases. In some instances, it may be equal to a time range of the predicate. For example if the event generator generates an event for the predicate *Happens(ic:S1:P(id1,a),t,ℜ(3,10))* the timestamp of the generated event would be *ℜ(3,10)*.

If a template from which a predicate has been generated, the *event generator* fetches the templates that depend on the generated events and feeds the generated event to those templates. This is because the events generated by the *event generator* may not have fixed time point as timestamp so they can not be treated as recorded events that are handled by the *event feeding* algorithm of Figure 4.9. Thus, to feed the derived events to the dependant templates the *event generator* checks if the derived event can be unified with the predicate and checks the quantification on the predicate time variable and the limits of the range. The possible cases considered by the *event generator* are shown in Table 4.2. The notation used in Table 4.2 has the same meaning as that of in Table 4.1, except the events. An event in Table 4.2 is shown as $P(a, (t_1,t_2))$, where *P* stands for the event signature, *a* represents the value of a variable in the event and (t_1,t_2) is the possible time range for the event to occur. The second row under the event column shows the checking of the event time range compliance with the predicate time range and other conditions that should be checked to update the truth-value of

a predicate. If the event time range intersects with the predicate time range, the *event generator* updates the truth-value of the predicate depending on the semantic of the quantifier of the predicate. This reasoning is described below. The *event generator* also checks the *pForm* condition of the predicate whose truth-value has been updated. If the event time range intersects with the predicate time range but the event time range is not within the predicate time range, then the predicate is marked as a *pForm* predicate (see definition 6), by setting *pForm* field of the predicate to *true*.

Existentially quantified predicates:

(i) A predicate of the form $(\exists t_p).P(\underline{x},t_p,[LB,UB])$ signifies there exist at least a single time point t_p in the range $[LB,UB]$ such that the evidence of predicate P can be found. The truth-value of a predicate of this form is set to *true* as soon as the first event that has intersecting time range with the time range *[LB, UB]* and that can be unified with P occurs (cases 1, 2, 9 and 10 in Table 4.2). The occurrence of an event that has intersecting time range with the range *[LB, UB]* but the event can not be unified (or can be unified, but the event is negated) with P, does not have any impact on the predicate (cases 3, 4, 11 and 12 in Table 4.2).

(ii) A predicate of the form $\neg (\exists t_p).P(\underline{x},t_p,[LB,UB])$ signifies there does not exist any single time point t_p in the range $[LB,UB]$ such that the evidence of predicate P can be found. The truth-value of a predicate of this form is set to *false* as soon as the first event that has intersecting time range with the time range *[LB, UB]* and that can be unified with P occurs (cases 5, 6, 13 and 14 in Table 4.2). The occurrence of an event that has intersecting time range with the range *[LB, UB]* but the event can not be unified (or can be unified, but the event is negated) with P, does not have any impact on the predicate (cases 7, 8, 15 and 16 in Table 4.2).

Universally quantified predicates:

(i) A predicate of the form $(\forall t_p).P(\underline{x},t_p,[LB,UB])$ signifies at any observable time point t_p in the range $[LB,UB]$ the evidence of the predicate P can be found. The truth-value of a predicate of the form $(\forall t_p).P(\underline{x},t_p,[LB,UB])$ is set to *true* if a derived event that has an intersecting time range with the time range *[LB, UB]* and can be unified with P occurs (see cases 17, 18, 25 and 26 in Table 4.2). A derived event that has intersecting time range with the range *[LB, UB]* but can not be unified with P or a derived event that can be unified, but the predicate is negated, does not have any impact on the predicate (cases 19, 20, 27 and 28 in Table 4.2).

Table 4.2: Possible predicate updating cases considered by the Event Generator

Predicate	**Deduced Event**			
	$P(a, [t_s,t_s])$	**$P(a, [t_s,t_e])$**	**$\neg P(a, [t_s,t_s])$**	**$\neg P(a, [t_s,t_e])$**
	Condition: $[t_s,t_s] \cap [LB, UB] \neq \emptyset$ && P.TV=UK	Condition: $[t_s,t_e] \cap [LB,UB] \neq \emptyset$ && P.TV=UK	Condition: $[t_s,t_s] \cap [LB,UB] \neq \emptyset$ && P.TV=UK	Condition: $[t_s,t_e] \cap [LB,UB] \neq \emptyset$ && P.TV=UK
$\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t	1 P.TV:=true X:=a $t_p := t_s$	2 P.TV:=true x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true	3 No Effect	4 No Effect
$\neg\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t $\forall t. \neg P(\underline{x},t_p,[LB,UB])$	5 P.TV:=false X:=a $t_p := t_s$	6 P.TV:=false x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true	7 No Effect	8 No Effect
$\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2	9 P.TV:=true X:=a $t_p := t_s$	10 P.TV:=true x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true	11 No Effect	12 No Effect
$\neg\exists t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2 $\forall t_p. \neg P(\underline{x},t_p,[LB,UB])$	13 P.TV:=false X:=a $t_p := t_s$	14 P.TV:=false x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true	15 No Effect	16 No Effect
$\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t	17 P.TV:=true X:=a $t_p := t_s$	18 P.TV:=true x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true	19 No Effect	20 No Effect
$\neg\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=t UB=t $\exists t_p. \neg P(\underline{x},t_p,[LB,UB])$	21 No Effect	22 No Effect	23 P.TV:=true x:=a $t_p := t_s$	24 P.TV:=true x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true
$\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2	25 P.TV:=true X:=a $t_p := t_s$	26 P.TV:=true x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true	27 No Effect	28 No Effect
$\neg\forall t_p.P(\underline{x},t_p,[LB,UB])$, LB=$t_1$ UB=t_2 $\exists t_p. \neg P(\underline{x},t_p,[LB,UB])$	29 No Effect	30 No Effect	31 P.TV:=true x:=a $t_p := t_s$	32 P.TV:=true x:=a $t_p := t_e$ **if** $[t_s,t_e] \not\in [LB,UB]$ P.pForm:=true

(ii) A predicate of the form $\neg(\forall t_p).P(\underline{x},t_p,[LB,UB])$ signifies not for all time point t_p in the range *[LB,UB]* the evidence of the predicate *P* can be found. The truth-value of a predicate of this form is set to *true* as soon as the first derived event that has intersecting time range with the time range *[LB, UB]* and that can not be unified (or can be unified, but the event is negated) with *P* occurs (cases 23, 24, 31 and 32 in Table 4.2). A derived event that has intersecting time range with the range *[LB, UB]*and that can be unified with *P*, does not have any impact on the predicate (cases 21, 22, 29 and 30 in Table 4.2).

4.4.3.2 The Algorithm

Figure 4.10 shows the algorithm for the *event generator*

```
Generate_event(T: template, M_temps: list of templates)
1.  if (each predicate P1 in the body of T has P1.TV=true and P1.SC=RE) then
2.    for each dependant DP in T do
3.      for each predicate P in head of T do
4.        if(DP.P unifies P)and(P.TV=UK or P.SC=NF)and(not partial(u_f|p) then
5.          P.TV:=true
6.          P.SC:=DE
7.          generate an event E
8.          E.SG:=P.SG
9.          E.NG:=P.NoQ
10          E.VB:=u_f|p
11.         if(T' is a Template in M_temps such that T'.Fid=DP.id) then
12.           for each predicate P' in T' do
13.             u_cur := imgu(E,P',u_f|P')
14.             ℜ(L,U):= ℜ(eval(P.LB),eval(P.UB))∩ ℜ(eval(P'.LB),eval(P'.UB))
15.              if(u_cur≠∅and (P'.TV=UK or P'.SC=NF)and(ℜ(L,U)≠∅) then
16.                 if(P'.Q=existential) then
                      /* dependent formula predicate: ∃t.p(x,t) or
                            ¬∃t.p(x,t) and derived predicate: p(a,t) */
17.                   if(E.NG=false) then
18.                     T" := T';
19.                     M_temps:= M_temps ∪ T"
20.                     P'.TV := ¬ P'NoQ
21.                     P'.SC := DE
22.                     if(ℜ(eval(P.LB),eval(P.UB)) is not within
                                              ℜ(eval(P'.LB),eval(P'.UB)) then
23.                           P'.pForm := true
24.                     end if
25.                     T'.updated := true
26.                     T'.U_f := T'.U_f ∪ u_cur
27.                     for each predicate P" in T' do
28.                       if P".UB.T_i is a time var in P".UB such as
                                                  P".UB.T_i.N=P'.TS.N then
29.                           P".UB.T_i.V := U.V
30.                       end if
31.                       if P".LB.T_i is a time var in P".LB such as
                                                  P".LB.T_i.N=P'.TS.N then
32.                           P".LB.T_i.V := U.V
33.                       end if
34.                     end for
35.                   end if  /* end of E.NG check */
36.                 else /* Universall predicate */
```

```
                    /* dependent formula predicate:∀t.p(x,t) or ¬∀t.p(x,t)*/
37.                   if(E.NG=P'.NoQ) then
38.                     T" := T';
39.                     M_temps:= M_temps ∪ T"
40.                     P'.TV := true
41.                     P'.SC := DE
42.                     if((ℜ(eval(P.LB),eval(P.UB)) is not within
                                                ℜ(eval(P'.LB),eval(P'.UB)) then
43.                           P'.pForm := true
44.                     end if
45.                     T'.updated := true
46.                     T'.U_f := T'.U_f ∪ u_cur
47.                     for each predicate P" in T' do
48.                        if P".UB.T_i is a time var in P".UB such as
                                                    P".UB.T_i.N=P'.TS.N then
49.                                 P".UB.T_i.V := U.V
50.                         end if
51.                         if P".LB.T_i is a time var in P".LB such as
                                                    P".LB.T_i.N=P'.TS.N then
52.                                    P".LB.T_i.V := U.V
53.                         end if
54.                      end for
55.                   end if  /* end of E.NG=P'.NoQ */
56.                 end if  /* end of  quantifier check */
57.               end if /* end of u_cur≠∅ */
58.             end for /* end of each p' */
59.             if T'.updated = true then check_consistency(T', M_temps)
60.           end if /* end of T'.Fid = DP.id */
61.         end if /* end of P.TV = UK */
62.       end for /* end of each p */
63.     end for /* end of each DP */
64. end if
end generate_event
```

Figure 4.10: Algorithm for the event generator

4.4.3.3 Explanation of the Algorithm

In line 1 the *event generator* checks if the truth-value of each predicate in the body of the template *T* is set to *true* using recorded events. The *for* loop starts in line 2 iterates for all the dependants *T* has. The *for* loop starts in line 3 iterates over each predicate *P* in the head of *T*. The *if* statement in line 4 checks the necessary conditions to generate an event. In the code segment between lines 5 and 10, truth-value of *P* is updated and a derived event *E* is generated. A target template *T'* that depends on *T* is picked up in line 11. The *for* loop starts in line 12 iterates over each predicate *P'* in *T'*. In line 13 the current most general unifier u_{cur} for *P'* and the event *E* is computed. If *E* can not be unified with *P'*, then u_{cur} would be empty. In line 14 the intersection of the time range of *E* and the time range of *P'* is calculated. The *if* condition in line 15 checks if *E* can be unified with *P'* and the truth-value of *P'* has already not been set by a recorded event. The *if* block between lines 16 and 36 considers the predicates with existentially quantified time variables, i.e. cases 1, 2, 5, 6, 9, 10, 13 and 14 in Table 4.2. A new instance of *T'* is created and added to the template list in lines 18 and 19 respectively. Truth-value and source of *P'* are set in lines 20 and 21 respectively. The

condition for *pForm* is checked and the predicate is marked as *pForm* between lines 22 and 24. The *for* loop between lines 27 and 34 updates the time range of other predicates in the template that depends on the time variable of P'. The *else* block between lines 36 and 56 considers the predicates with universally quantified time variables, i.e. cases 17, 18, 23, 24, 25, 26, 31 and 32 in Table 4.2. A new instance of T' is created and added to the template list in lines 38 and 39 respectively. Truth-value and source of P' are set in lines 40 and 41 respectively. The condition for *pForm* is checked and the predicate is marked as *pForm* between lines 42 and 44. The *for* loop between lines 47 and 54 updates the time range of other predicates in the template that depends on the time variable of P'. The if condition in line 59 checks if T' has been updated, in such case consistency checker is called to check T' for possible violation.

4.4.4 The Consistency Checker

The consistency checker receives a template and makes a decision about the template if none of the predicates in the template has the status *UK* and updates the monitoring status of the template. The algorithm used to implement the *Consistency Checker* is described below.

4.4.4.1 Overview of the Algorithm for the Consistency Checker

The *consistency checker* makes monitoring decision according to the rules described below. These rules are based on the formal definitions of the property deviation types presented in Section 4.3.

- If the *truth value* (*TV*)of all the predicates in the template is set to *true*, the template (formula instance) is satisfied.
- A template that signifies a formula instance is inconsistent with the recorded behaviour of the system if,
  - (i) the *truth value* (*TV*) of all the predicates in the body of the template is set to *true*
  - (ii) the *truth value* (*TV*) of at least one predicate in the head is set to *false*
  - (iii) the *source of event* (*SC*) of none of the predicates in the template is set to *DE*,

  According to Definition 3 in Section 4.3, a formula f: $C \Rightarrow A$ is inconsistent with the recorded behaviour, if $\neg f$ (i.e. $C \wedge \neg A$) can be entailed by the recorded behaviour of the system. The condition (i) establishes C, condition (ii) establishes $\neg A$ and condition (iii) guarantees that only recorded events are used to detect the inconsistency.

- A template that signifies a formula instance is inconsistent with the expected behaviour of the system if,
  - (i) the *truth value* (*TV*) of all the predicates in the body of the formula is set to *true*
  - (ii) the *truth value* (*TV*) of at least one predicate in the head is set to *false*
  - (iii) the *source of event* (*SC*) of at least one predicate in the template is set to *DE*,

  According to Definition 4 in Section 4.3, a formula $f: C \Rightarrow A$ is inconsistent with the expected behaviour, if $\neg f$ (i.e. $C \wedge \neg A$) can be entailed by the recorded behaviour and the expected behaviour of the system. The condition (i) establishes C, condition (ii) establishes $\neg A$ and condition (iii) guarantees that derived event (expected behaviour) is used to establishe the truth value of at least one predicate in the formual to detect the inconsistency.

- A template, that signifies an instance of a behavioural property, shows unjustified behaviour if,
  - (i) the *truth value* (*TV*) of all predicates in the head of the template is set to *true*
  - (ii) the *source of event* (*SC*) of all the predicates in the head of the template is set to *RE*,
  - (iii) there is another template T' that signifies a different instance of the same behavioural property and T' is satisfied by recorded events only.
  - (iv) the *truth value* (*TV*) of at least one predicate in the body is set to *false* and the *source of event* (*SC*) for this predicate is set to *DE*,

  According to Definition 5 in Section 4.3, a behavioural property $f: C \Rightarrow A$ shows unjustified behaviour, if f is satisfied by the recorded behaviour of the system, but there is at least one condition in the body of f which would not be satisfied by the expected behaviour of the system. The conditions (i)–(iii) ensure that the behavioural property is satisfied by the recorded behaviour of the system and, condition (iv) establishes that there is at least one condition in the body of the template that is not satisfied by the derived events.

- A template that signifies a formula instance is possibly inconsistent with the expected behaviour of the system if,
  - (i) at least one predicate in the template has *pForm* set to *true*,
  - (ii) the *truth value* (*TV*) of all the predicates in the body of the formula is set to *true*
  - (iii) the status of at least one predicate in the head is set to *false*
  - (iv) the *source of event* (*SC*) of at least one predicate in the template is set to *DE*

According to Definition 6 in Section 4.3, a formula $f: C \Rightarrow A$ is possibly inconsistent with the expected behaviour, if $\neg pForm(f)$ (i.e. $pForm(C) \wedge \neg pForm(A)$) can be entailed by the recorded behaviour and the expected behaviour of the system. The condition (i) ensures that the *pForm* of the template is used, condition (ii) establishes $pForm(C)$, condition (iii) establishes $\neg pForm(A)$ and condition (iv) guarantees that derived event (expected behaviour) is used to establishe the truth value of at least one predicate in the formual to detect the inconsistency.

- A template, that signifies an instance of a behavioural property, shows potentially unjustified behaviour if,
  - (i) at least one predicate in the template has *pForm* set to *true*,
  - (ii) the *truth value* (*TV*) of all predicates in the head of the template is set to *true*
  - (iii) the *source of event* (*SC*) of all the predicates in the head of the template is set to *RE*,
  - (iv) there is another template *T'* that signifies a different instance of the same behavioural property and *T'* is satisfied by recorded events only.
  - (v) the *truth value* (*TV*) of at least one predicate in the body is set to *false* and the *source of event* (*SC*) for this predicate is set to *DE*,

According to Definition 7 in Section 4.3, a behavioural property $f: C \Rightarrow A$ shows potentially unjustified behaviour, if f is satisfied by the recorded behaviour of the system, but there is at least one condition in the body of $pForm(f)$ which would not be satisfied by the expected behaviour of the system. The condition (i) ensures that *pForm* of f is used, conditions (ii)–(iv) ensure that the behavioural property is satisfied by the recorded behaviour of the system and, condition (v) establishes that there is at least one condition in the body of the template that is not satisfied by the derived events.

4.4.4.2 The Algorithm

Figure 4.11 shows the algorithm for the consistency checker,

```
check_consistency(T: formula template, Templates: list of all templates)
1. Let T is the received Template and Templates is the list of all templates
2. if T.forChecking=true and there is no Predicate P in T that has P.TV=UK
                                                                    then
3.      if (each predicate P in T has P.TV=true) then
4.              T.ST = SAT
5.      else if (at least one Predicate P in head of T has P.TV=false) and
          (each predicate P in body of T has P.TV=true) and (no predicate P
            in T has P.SC=DE) then
```

```
6.                  T.ST = I_R_B
7.      else if (at least one Predicate P in head of T has P.TV=false) and
              (each  predicate  P in body of T has P.TV=true) and (at least
              one predicate P in T has P.SC=DE) and (no predicate P  has
              P.pForm=true) then
8.                  T.ST = I_E_B
9.      else if (each Predicate P in head of T have P.TV=true and P.SC=RE)
             and (at least one Predicate P in body of T has P.TV=false and
             P.SC=DE) and (no predicate P has P.pForm=true) and (T' is a
             template in Templates such that T'.Fid = T.Fid and T'.ST=SAT and
             each predicate P' in T' has P'.SC=RE) then
10.                 T.ST = U_B
11.      else if (at least one Predicate P in head of T has P.TV=false) and
              (each predicate P in body of T has P.TV=true) and (at least one
              predicate P  has  P.pForm=true) then
12.                 T.ST = P_I_E_B
13.     else if (each Predicate P in head of T have P.TV=true and P.SC=RE)
             and (at least one Predicate P in body of T has P.TV=false and
             P.SC=DE) and (at least one predicate P has P.pForm=true) (T'
             is a template in Templates such that T'.Fid = T.Fid and
             T'.ST=SAT and  each predicate P' in T' has P'.SC=RE) then
14.                 T.ST = P_U_B
15.     end if
16. end if
end check_consistency
```

Figure 4.11: Algorithm for the consistency checker

4.4.4.3 Explanation of the Algorithm

In line 1 the *consistency checker* receives a Template *T*. In line 2 a check has been made to make sure that *T* signifies a formula instantiation that should be checked (i.e. *T* is not template of an assumption) and all the predicates in *T* have truth-value set. In line 3 conditions for formula satisfiability is checked and if these conditions are true the status *T* is set to *SAT*, which stands for *Satisfied*, in line 4. In line 5 conditions for *inconsistency of recorded behaviour* are checked. If these conditions are true the status of *T* is set to I_R_B, which stands for *inconsistency of recorded behaviour*, in line 6. In line 7 conditions for *inconsistency of expected behaviour* are checked and if these conditions are true the status of *T* is set to I_E_B, which stands for *inconsistency of expected behaviour*, in line 8. In line 9 conditions for *unjustified behaviour* are checked. If these conditions are true the status of *T* is set to U_B, which stands for *unjustified behaviour*, in line 10. In line 11 conditions for *potential inconsistency of expected behaviour* are checked and if these conditions are true the status of *T* is set to P_I_E_B, which stands for *potential inconsistency of expected behaviour*, in line 12. In line 13 conditions for *potentially unjustified behaviour* are checked. If these conditions are true the status of *T* is set to P_U_B. which stands for *potentially unjustified behaviour*, in line 14.

4.4.5 Analysis of the Monitoring Algorithm

In this section we analyse the monitoring algorithm presented above. First we summarize some limitations of this algorithm and then present a formal analysis of it.

The limitations of the monitoring scheme are listed below,

- The supports only checking of future formulas. It does not support past formula.
- The *event generator* applies only deductive reasoning and does not apply abductive reasoning to derive the possible behaviour of the system.
- The scheme generates monitoring decision only at the formula instance level, not at the formula level. It should however, be appreciated that the monitoring decision at the formula level can be established quite easily from the monitoring decisions at the formula instance level. For example, consider the following formula,

(∀ t1:Time) (∀t2:Time)
Happens(in:IS:FindAvailableCar(_oID1, _pID), t1, ℜ(0,20)) ∧
Happens(ir:IS:FindAvailableCar(_oID1),t2, ℜ(t1,t2)) ⇒ oc:self:sub(t2, t1) < 5

It specifies for all observable time points *t1* in [0,...,20], and other time points *t2* after *t1* the formula must be satisfied. The formula is satisfied if all the formula instances (templates) generated by the monitor in the specified time range are satisfied, and the formula is violated if at least one formula instance (template) generated by the monitor in the specified time range is violated. The monitoring algorithm as it stands does not report collective result for the formula.

In the rest of this section we present the formal analysis of the monitoring algorithm.

4.4.5.1 Soundness

The soundness of the *monitoring algorithm* can also be established by explaining the soundness of the *event feeder algorithm*, the *event generator algorithm* and the *consistency checker algorithm*. In the following we establish the soundness of the *event feeder algorithm*, the *event generator algorithm* and the *consistency checker algorithm*.

Theorem 4.1 (*soundness of the event feeder algorithm*)

Given a template instance *T* and an event *e*, the *feed* method updates the truth value of a predicate *P* in *T* if and only if *P* and *e* conforms to one of the predicate updating cases presented in Table 4.1.

Proof:

The *event feeder algorithm* updates the truth value of a predicate *P* in a template instance *T* by applying the principle negation as failure or using a recorded event *e*. In either case it covers all the predicate truth value updating cases described in Table 4.1. This is because:

(i) The *event feeder algorithm* updates the truth value of a predicate *P* in a template instance *T* using a recorded event *e* only if *e* can be unified with *P* and *P.LB* <= *e.TS* <= *P.UB*. This covers the cases 1, 4, 7, 10, 13, 16, 19 and 22 in Table 4.1 (see Section 4.4.2.3 for the explanation of the algorithm). The updates performed in these cases are sound.

(ii) The *event feeder algorithm* updates the truth value of a predicate *P* in a template instance *T* by applying the principle negation as failure only if all the variables in *P* are bound to some value (that ensures there was a possibility of having a recorded event to update the truth value of *P* given the previous recorded events that are used to update the truth value of other predicates in *T*) and it is guaranteed that given *e* there is no possibility of having a recorded event that can be used to update the truth value of *P* in *T*. More specifically, the principle negation as failure is applied if (a) *P.TS* is universally quantified and *e* can be unified with *P* and *e* is negated or if *e* can not be unified with *P*. This covers the cases 17 and 23 in the Table 4.1, and (b) *e.TS* > *P.UB* that is in cases 3, 6, 9, 12, 15, 18, 21 and 24 in Table 4.1 (see Section 4.4.2.3 for the explanation of the algorithm). The updates performed in these cases are sound.

Since in any case other than the two cases described above the truth value of a predicate is not updated by the *event feeder algorithm* and Table 4.1 exhausts all possibilities of event/predicate combinations, it is ensured that no anomaly can be introduced in the monitoring process by the *event feeder algorithm*.

Theorem 4.2 (*soundness of the event generator algorithm*)

Given a template instance *T* the *generate_event* method (i) generates an event *P* from *T* if and only if *T* satisfies the conditions to generate an event and (ii) updates the truth value of a predicate *P'* in a template instance *T'* if and only if *T'* depends on *T*, *P* and *P'* conforms to one of the predicate updating cases presented in Table 4.2.

Proof:

The *event generator algorithm* derives an event P from a template instance T only if T satisfied the conditions presented in Section 4.4.3.1 (see Section 4.4.3.3 for the explanation of the algorithm) which are the sufficient conditions to generate an event and update a template instance T' that depends on T. The *event generator algorithm* updates the truth value of a predicate P' in T' using P only if P can be completely unified with P' and [P.LB, P.UB]$\cap$[P'.LB, P'.UB] $\neq \varnothing$. This ensures that all the cases presented in Table 4.2 are considered by the *event generator algorithm* (see Section 4.4.3.3 for the explanation of the algorithm).

Theorem 4.3 (*soundness of the consistency checker algorithm*)

Given a template instance T the *check_consistency* method updates the status of T if and only if T complies with the formal definition of one of the inconsistency type presented in Section 4.3.

Proof:

The *consistency checker algorithm* updates the status of a template instance T only if T satisfied one of the conditions described in Section 4.4.4.1 which are the conditions based on the formal definitions of the inconsistencies presented in Section 4.3 (see Section 4.4.4.3 for the explanation of the algorithm).

4.4.5.2 Completeness

The *monitoring algorithm* (see Figure 4.8) is complete. This is because it considers each single event e in the event database and for each single event it considers each single template instance T in the formula database to feed the event to the template instance using the *event feeder algorithm* presented in Figure 4.9.

The *event feeder algorithm* considers each single predicate P in the template instance to check if the template instance should be updated by the event. The *event feeder algorithm* performs an exhaustive check to ensure the time compliance and covers all the possible cases presented in Table 4.1 as it is shown in the explanation of the algorithm in Section 4.4.2.3. New instances of templates are created by the *event feeder algorithm,* if a new event can be unified with an unconstrained predicate of a template, or the variable bindings of the predicate have values that are different from the event variable bindings values. In this case, all partially instantiated templates of formulas which do not bind the variables of the predicate that can be

unified with the event will be updated. This covers the possibility of having different template instances of the same formula because of different variable bindings.

The truth-value of a predicate is updated by the *event feeder algorithm* by applying the principle negation as failure only if all the variables in the predicate are bound to some value (this ensures that a recorded event to update the predicate should have occurred given other recorded events that were used to update the truth value of other predicates in the template) and the current event time exceeds the upper boundary of the predicate (this ensures that it is not possible to have any more recorded event whose time stamp would be within the time range of the predicate). The *event feeder algorithm* marks a template instance to be deleted only if the time stamp of the current event exceeds the upper boundary of a predicate with an undefined truth value and at least one variable which is not bound to some value. These two conditions guarantee that it would not be possible to establish the truth value of the predicate either by a future recorded event or by the principle negation as failure.

The *monitoring algorithm* uses the *event generator* algorithm presented in Figure 4.10 to feed the derived events to template instances. Like the *event feeder algorithm* the *event generator algorithm* also considers each single predicate P in a template instance T to check if the template instance should be updated by the derived event. Similarly to the *event feeder algorithm* the *event generator algorithm* performs an exhaustive check to ensure that the time range of the event is within the time range of the predicate and covers all the possible cases for derived events presented in Table 4.2 as it is shown in the explanation of the algorithm in Section 4.4.3.3. If the *event generator algorithm* updates the truth value of a predicate in a template using a derived event, it also creates a new instance of the template to cover the possibility of updating the same template and predicate using recorded events or another derived event.

The *monitoring algorithm* uses the *consistency checker algorithm* presented in Figure 4.11 to detect violations in template instances. The *consistency checker algorithm* is also complete as it checks the template instance for each single type of inconsistency defined in Section 4.3 as explained in Section 4.4.4.1.

4.4.5.3 Complexity

The complexity of the *monitoring algorithm* is determined by the complexity of the *event feeder algorithm*, the *event generation algorithm* and the *consistency checker algorithm*. In

the following we analyse the complexity of these algorithms. In the complexity calculation we assume each template has a maximum n number of predicates and each predicate has a maximum m number of variables.

Complexity of the Event feeder algorithm:

In the algorithm presented in Figure 4.9, the statements between lines 4 and 152 inside the *for* loop in line 2 can be divided into 7 blocks such that one of these blocks will be executed for each predicate in the template, i.e. for each iteration of the *for* loop in line 2. These blocks are, *Block 1* (*B1*) between lines 4 –24, *Block 2* (*B2*) between lines 27 – 44, *Block 3* (*B3*) between lines 46 – 62, *Block 4* (*B4*) between lines 66 – 88, *Block 5* (*B5*) between lines 88 – 110, *Block 6* (*B6*) between lines 110 – 130, *Block 7* (*B7*) between lines 131 – 152.

There are some statements common among the blocks, we define these statements as, *Template Creation Statements* (*TCS*) e.g. between lines 6 and 9 or between lines 28 and 31, *Time Range Update Statements* (*TRS*) e.g. between lines 14 and 21 or between lines 100 and 107. Before computing the complexity of the blocks *B1–B*. we compute the complexity of *TCS* and *TRS* that will be used to compute the complexity of *B1– B7*. The complexity of a *TCS* is $O(nm)$. For clarification consider the *TCS* between lines 6 and 9. The complexity of the condition checking in line 6, which requires to check the value of m number of variables, is $O(m)$, the complexity of creating new template instance is $O(nm)$, which involves copying of n number of predicates from the old template instance to the new template instance and each predicate may have m number of variables to be copied from the old template instance to the new template instance. The complexity of a *TRS* is $O(n)$. For clarification consider the *TRS* between lines 14 and 21. The complexity of the condition checking and value updating in lines 15 and 16 respectively (and lines 18 and 19 respectively) is $O(1)$ and this is repeated for n number of predicates in the template.

With the computed complexity of *TCS* and *TRS* we compute the complexity of each single block and then the complexity of the whole *event feeder algorithm.*

For *B1*, the complexity of the checking in line 5 is $O(1)$. The complexity of the *TCS* between lines 6 and 9 is $O(nm)$. Each of the value updating statement between lines 10 and 13 contributes $O(1)$ to the overall complexity. The complexity of the *TRS* between lines 14 and 21 is $O(n)$

Hence the complexity of *B1* is, $O(1) + O(nm) + O(1) + O(n) \approx O(nm)$

For *B2*, the complexity of the checking in line 27 is $O(1)$. The complexity of the *TCS* between lines 28 and 31 is $O(nm)$.Each of the value updating statement between lines 32 and 35 contributes $O(1)$ to the overall complexity. The complexity of the *TRS* between lines 36 and 43 is $O(n)$

Hence the complexity of *B2* is, $O(1) + O(nm) + O(1) + O(n) \approx O(nm)$

For *B3*, the complexity of the checking in line 46 is $O(m)$ which requires to check the value of m number of variables. The complexity of each of the condition checking and value updating statement between lines 47 and 49 is $O(1)$. The complexity of the *TRS* between lines 50 and 57 is $O(n)$

Hence the complexity of *B3* is, $O(m) + O(1) + O(n) \approx O(n)$

For *B4*, the complexity of the checking in line 66 and 67 is $O(1)$. The complexity of the *TCS* between lines 68 and 71 is $O(nm)$.Each of the value updating statement between lines 72 and 75 contributes $O(1)$ to the overall complexity. The complexity of the *TRS* between lines 76 and 83 is $O(n)$. The complexity of line 85 is $O(1)$.

Hence the complexity of *B4* is, $O(1) + O(nm) + O(1) + O(n) \approx O(nm)$

For *B5*, the complexity of the checking in line 89 is $O(1)$. The complexity of the *TCS* between lines 92 and 95 is $O(nm)$.Each of the value updating statement between lines 96 and 99 contributes $O(1)$ to the overall complexity. The complexity of the *TRS* between lines 100 and 107 is $O(n)$

Hence the complexity of *B5* is, $O(1) + O(nm) + O(1) + O(n) \approx O(nm)$

For *B6*, the complexity of the checking in lines 112 and 115 are $O(1)$ and $O(m)$ respectively. The complexity of the value updating statements between lines 116 and 118 is $O(1)$. The complexity of the *TRS* between lines 119 and 126 is $O(n)$

Hence the complexity of *B6* is, $O(1) + O(m) + O(n) \approx O(n)$

For *B9*, the complexity of the checking in line 132 is $O(m)$ which requires to check the value of m number of variables. The complexity of each of the condition checking and value updating statement between lines 133 and 139 is $O(1)$. The complexity of the *TRS* between lines 140 and 147 is $O(n)$

Hence the complexity of *B9* is, $O(m) + O(1) + O(n) \approx O(n)$

The *for* loop in line 2 of the *event feeder algorithm* iterates for n number of predicates, and for each iteration one of the blocks (*B1 – B9*) is executed. Applying the complexity of the blocks computed above, the worst case complexity of the *event feeder algorithm is,* $O(n^2m)$.

Complexity of the event generator algorithm:

In the algorithm presented in Figure 4.10, the statements between lines 15 and 57 in the *event generator algorithm* can be divided into two blocks such that one of these blocks will be executed if a derived event is generated by the event generator. These blocks are *Block 1* (*B1*) between lines 15 and 35, and the *Block 2* (*B2*) between lines 37 and 55. It should be noted that both *B1* and *B2* will have similar complexity as they have similar type of statements under different conditions. To compute the complexity of the whole algorithm we apply a bottom up approach, i.e. we first compute the complexity of *B1* (*B2* has similar complexity), then we compute the complexity of the *for* loop in line 12, then the complexity of the for loop in line 3 and in line 2 respectively.

For *B1*, the complexity of the condition checking in line 15, 16 and 17 is $O(1)$. The statement in line 18, that creates a new template instance has complexity $O(nm)$, which involves copying of n number of predicates from the old template instance to the new template instance and each predicate may have m number of variables to be copied from the old template instance to the new template instance. Each of the value updating statement between lines 19 and 26 contributes $O(1)$ to the overall complexity. The complexity of the *TRS* between lines 27 and 35 is $O(n)$

Hence the complexity of *B1* (also for *B2*) is, $O(1) + O(nm) + O(1) + O(n) \approx O(nm)$

The *for* loop in line 12, repeats for n number of predicates in the template and for each iteration either *B1* or *B2* is executed. Hence the complexity of the *for* loop in line 12 (the statements between lines 12 and 58), is $O(n^2m)$.

The *for* loop in line 3 repeats for n number of predicates in the template head and for each iteration it executes the statements between lines 4 and 62. Each of the value updating statement between lines 4 and 11 contributes $O(1)$ to the overall complexity of this *for* loop. The *for* loop in line 12 has complexity $O(n^2m)$ and the complexity of the consistency checker invoked in line 59 is $O(n)$ (see below). Hence the complexity of the *for* loop in line 3 (the statements between lines 3 and 62), is $O(n^3m)$.

The *for* loop in line 2, repeats for n number of dependants, and in each iteration the *for* loop in line 3 is executed which has complexity $O(n^3m)$. Hence the complexity of the *for* loop in line 2 (the statements between lines 2 and 63), is $O(n^4m)$.

The complexity of the condition checking in line 1 is $O(n)$, since it checks the truth-value of n number of predicates in the template body.

Thus the overall complexity of the *event generator algorithm* is, $O(n) + O(n^4m) \approx O(n^4m)$

Complexity of the Consistency Checker:

In the algorithm presented in Figure 4.11, each of the condition checking statements in lines 2, 3, 5, 7, 9, 11 and 13 has complexity $O(n)$ as each condition checks the truth value of n number of predicates in the template. Each of the value updating statements in lines 4, 6, 8, 10, 12 and 14 has complexity $O(1)$.

Thus the overall complexity of the *consistency checker algorithm* is, $O(1) + O(n) \approx O(n)$

Complexity of the Monitoring algorithm:

In the algorithm presented in Figure 4.8, the complexity of each of the statements in lines 1, 2, and 3 is $O(1)$. The complexity of the *for* loop between lines 4 and 8 is $O(n)$ as it creates empty template for n number of formulas. The complexity of each value updating statements in lines 9 –12 or 18–22 is $O(1)$.

The *monitoring algorithm* picks an event in line 14 and processes the event through the *for* loop in line 29. We compute the complexity of processing all the events when , (i) monitoring is performed with respect to recorded events only and (ii) monitoring is performed with respect to recorded and derived events.

Let F_n is the number of initial template instances and E_n is the number of events to be processed.

Case 1 (monitoring with respect to recorded events only)

The *for* loop in line 29 repeats for each template instance and each time it invokes the *event feeder algorithm* in line 30 to feed the event to a template instance and it invokes the *consistency checker algorithm* in line 33 to check the consistency of a template instance. The complexity of the *event feeder algorithm* and the *consistency checker* algorithm are $O(n^2m)$ and $O(n)$ respectively.

The number of template instances to be considered for event E_1 is F_n

Complexity to process E_1 is $F_n * O(n^2m)$

In the worst case the event E_1 will be unified with each of the F_n template instances and for each of these instances the *event feeder algorithm* will create a new template instance without deleting any instance. Therefore the number of template instances to be considered for event E_2 is, $F_n + F_n = 2F_n$

Thus the complexity to process E_1 and E_2 is $(F_n + 2F_n) * O(n^2m)$

At the next step, event E_2 can also be unified with each of the $2F_n$ template instances and for each template instance the *event feeder algorithm* will create a new template instance and no template instance will be deleted. Therefore, the number of template instances to be considered for event E_3 will be, $2F_n + 2F_n = 4F_n$

Thus complexity to process E_1, E_2 and E_3 is $(F_n + 2F_n + 4F_n) * O(n^2m)$

By applying similar reasoning, the complexity to process all the E_n events,

$$(F_n + 2F_n + 2^2 F_n + 2^3 F_n + \ldots\ldots + 2^{En-1} F_n) * O(n^2m)$$
$$= F_n (1 + 2 + 2^2 + 2^3 + \ldots\ldots + 2^{En-1}) * O(n^2m)$$
$$= F_n * (2^{En} - 1) * O(n^2m)$$

Case 2 (monitoring with respect to recorded events and derived events)

The *for* loop in line 29 repeats for each template instance and each time it invokes the *event feeder algorithm* in line 30 to feed the event to a template instance, it invokes the *consistency checker algorithm* in line 33 to check the consistency of a template instance and it invokes the *event generator* in line 35 to generate derived events from the template instance. The complexity of the *event feeder algorithm,* the *consistency checker* algorithm and the *event generator* algorithms are $O(n^2m)$, $O(n)$ and $O(n^4m)$ respectively.

The number of template instances to be considered for event E_1 is F_n

The complexity to process E_1 is $F_n * O(n^4m)$

In the worst case the event E_1 will be unified with each of the F_n template instance and from each template instance the *event feeder algorithm* will create a new template instance. Also in the worst case, derived events can be generated from each template instance and the *event generator* creates a new template instance from each template instance. Therefore the number of template instances to be considered for event E_2 is, $F_n + F_n + F_n = 3F_n$

Complexity to process E_1 and E_2 is, $(F_n + 3F_n) * O(n^4m)$

In the worst case the event E_2 can be fed to each of the $3F_n$ template instance and from each template instance the *event feeder algorithm* creates a new template instance. Also in the worst case derived events can be generated from each template instance and the *event generator* creates a new template instance from each template instance. Therefore the number of template instances to be considered for event E_3 is, $3F_n + 3F_n + 3F_n = 9F_n$

Complexity to process E_1, E_2 and E_3 is $(F_n + 3F_n + 9F_n) * O(n^4m)$

By applying similar reasoning,

Complexity to process all the E_n events,

$$(F_n + 3F_n + 3^2F_n + 3^3F_n + \ldots\ldots + 3^{En-1}F_n) * O(n^4m)$$
$$= F_n(1 + 3 + 3^2 + 3^3 + \ldots\ldots + 3^{En-1}) * O(n^4m)$$
$$= F_n * \frac{3^{En} - 1}{2} * O(n^4m)$$

Chapter Five

Implementation of the Monitoring Framework

5.1 Overview

In this chapter we discuss the implementation of the monitoring framework presented in Chapter 2. More specifically in, Section 5.2 we present the implementation architecture of the framework and discuss the design choices and different components of this architecture along with their implementation. Subsequently in section 5.3, we describe a prototype of the monitoring framework and present a scenario of interacting with this prototype in order to demonstrate how it could be used by a human actor.

5.2 Implementation Architecture

For the convenient of the reader we show the implementation architecture of the monitoring framework again in Figure 5.1. As shown in this figure this architecture incorporates eight main components. These components are: a *behavioural properties extractor*, an *event receiver*, a *monitor manager*, a *monitor*, a *monitoring console*, a *simulator,* an *event database handler* and a *formula database handler*. This figure also shows the names of the interfaces exposed by the components of the architecture.

In this section we first discuss the major design choices and implementation issues of the architecture. We then discuss the mechanisms used to realise the functionality of different components of the architecture. We also describe the interfaces exposed by different components in the architecture along with necessary data structure. The primitive data types are presented as *xsd:data_type* where the prefix *xsd* signifies the XSD data types specification [Xml04b] and *data_type* signfies the name of the data type. The complex data types are defined using the primitive data types.

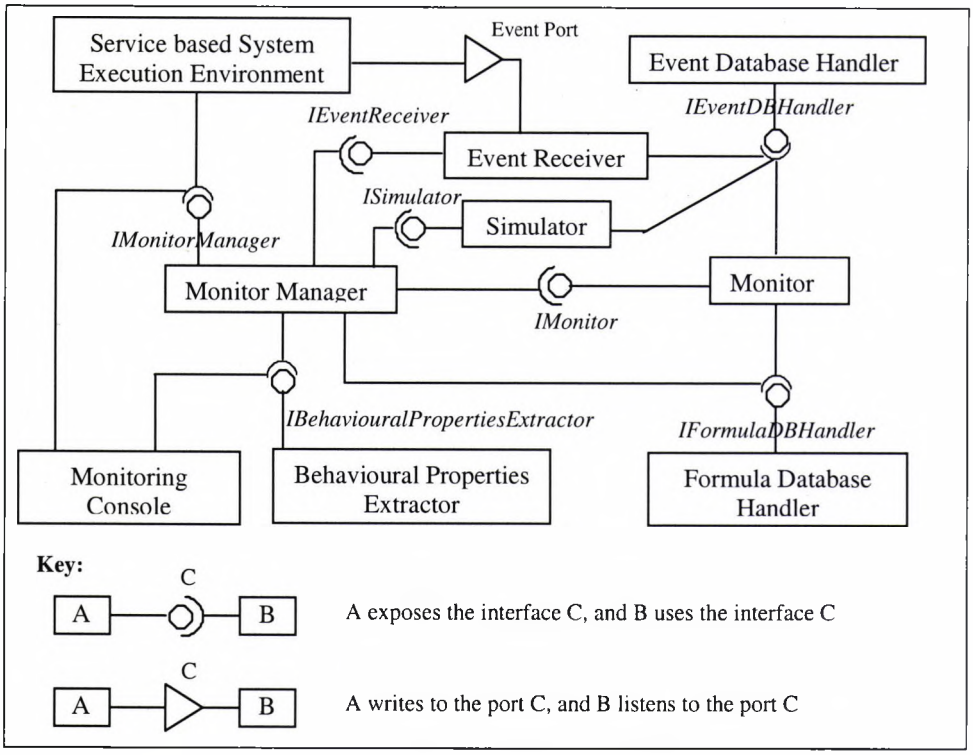


Figure 5.1: Implementation architecture of the monitoring framework

5.2.1 Design Choices and Implementation Issues

In the following we describe the major design criteria adopted and the implementation issues considered during the development of the architecture for our monitoring framework.

5.2.1.1 Selection of Web Service Composition Language

As discussed in Chapter 2, most of the web service orchestration and web service choreography languages, including BPEL, OWL-S, WSCI and WS-CDL, overlap in many aspects, and it is not easy to pick up a single language. Since this research focuses on the runtime requirements monitoring of web service based systems orchestrated by some executable workflow, WSCI and WS-CDL were not considered as these standards can be used to express non executable choreographies of web services. Thus, the remaining contesters were BPEL and OWL-S. Our selection was BPEL because of its wider acceptance by the industry. More importantly BPEL has been voted as a standard service composition language by OASIS [Fos06]. Another reason for choosing BPEL over OWL-S was the availability of tool support for it as discussed in Chapter 2.

5.2.1.2 Selection of BPEL Execution Engine

Because of its wide acceptance, in industry and academia, a number of tools are now available to support the design and/or execution of BPEL process, including BPWS4J from IBM [Bpw03], Oracle BPEL Process Manager from Oracle [Ora04], and ActiveBPEL from Active BPEL LLC [Act05]. To select an engine that best support our objectives, we considered how easily we could generate run time events without hampering the normal execution of the engine. Bpws4j uses log4j [Log03] to generate the execution log of a BPEL process and the logging information contains all the execution details of the BPEL process. The Oracle BPEL Process Manager also uses log4j to generate the execution log of a BPEL process but the logging information does not contain the execution details of the BPEL process. One possibility with Oracle BPEL Process Manager is to generate BPEL process related events by capturing all the SOAP messages being exchanged between the BPEL process and its constituent services. But in this approach it is not possible to capture the internal states of the BPEL process as for example the values of the internal variables of a BPEL process. ActiveBPEL is an open source engine that can be instrumented to generate all the required events for monitoring. This however would require some extra effort. Considering all these issues we decided that bpws4j was the engine that best fitted with the purpose of our research.

5.2.1.3 Modularity

The architecture presented in Figure 5.1, is composed of independent components that exchange data inside the framework. This ensures the applicability of the architecture to different contexts. To be more specific the *monitor* is the component in our architecture that, given a set of properties to be verified and a sequence of events, checks the compliancy of the recorded and expected events against the properties. The separation of the *monitor* from the other components of the architecture enables the *monitor* to be used as a component or as a web service. The *event receiver* is the component that receives run time events from the BPEL process execution engine. Although in our implementation we have selected bpws4j as the BPEL process execution engine, the architecture of our framework makes it applicable to any BPEL process execution engine by developing an *event receiver* that can capture events from this engine. Similarly the *behavioural property extractor* is the component that extracts behavioural properties and identifies monitorable events from BPEL process specification. Note, however, that the architecture of Figure 5.1 is applicable to any other web service flow specification language by developing a *behavioural property extractor* that could extract

behavioural properties and identify monitorable events from a web service composition process specified in this.

5.2.1.4 Data Storage

Storage for the run time events and the formula instances is an important issue to be considered. The use of main memory to store events and formula instances would significantly increase the efficiency of the monitoring process by reducing the data access time. Nonetheless, the use of main memory for data storage imposes restriction on the scalability of the framework as the number of events and the number of formula to be monitored in a complex system could be unlimited. This dictates us to use secondary memory for the storage of run time events and formula instances. Again in case of secondary storage, the use of a dedicated database management system (DBMS), like mysql [Mys95] or Oracle [Ora06] would increase the data access time hence degrade the performance of the monitoring process. Considering all these, we decided to implement the run time storage of events and formula instances using the file system of the machine used to run our framework.

5.2.2 Behavioural Properties Extractor

The *behavioural properties extractor* extracts the properties to be monitored from the specification of the composition process of service based systems expressed in BPEL. This component also allows to save formulas in XML and import formulas from XML file.

In this section we first overview BPEL and then describe the patterns used to transform BPEL specification into EC. We also discuss the mechanism used to extract the behavioural properties from BPEL specification with the help of an illustrative example. The behavioural properties extractor offers its functionality through *IbehaviouralPropertiesExtractor* interface. We describe the *IbehaviouralPropertiesExtractor* interface at the end of this section.

5.2.2.1 Overview of BPEL

BPEL is an XML based language to specify executable business processes, which deploy web-services to achieve their goals. The basic structure of a BPEL process specification is shown in Figure 5.2 [Bpe03]. The major elements of a BPEL process specification are:

Process: This is the root element that marks the start of a process definition.

Partners: Partner elements in a BPEL process specification specify the list of partners of a process, i.e. the web services that interact with the process. This is a mechanism to describe how the services (i.e. partners), which the process interacts with, are interrelated and the roles played by the services. This is done through service linking. Service links are used to specify the relationship of two services, the roles of each service in the relationship and the interface that each service provides. Usually service link types are defined in service's WSDL document.

Variables: This element holds a list of internal variables that of the BPEL process. A process variable is a typed data structure, where the type of a variable may be a WSDL message type or an XML schema simple type. Variables store the contents of messages that are exchanged between the process and its constituent web-services for a specific instance of the process, or store data that related to the state of the process but not exchanged between the partners.

CorrelationSets: This element holds a list of correlation sets that enables the process to keep track of the state of each process instance. A correlation set contains a number of message properties which when taken together form a unique key that can be used to distinguish that message from all other instances of that message from other process instances. This unique key helps the process executor to identify a particular process instance. Like service link type, correlation set, i.e. which properties of a message should form a unique key, is declared in service's WSDL document.

FaultHandlers: This element holds a list of elements that enable a process to handle faults found within the process. A *catch* element inside the *faultHandlers* is used to handle a specific fault within the process, and a *catchAll* element inside the *faultHandlers* is used to handle any fault not handled by a more specific *catch* element. If *catch* or *catchAll* element captures a fault, it performs a predefined action defined by *activity* (see below).

EventHandlers: This element contains a list of event and action pair. On the occurrence of any event from the list associated action is performed. The events can be of two types, (i) *message event* which signifies the arrival of a message, and (ii) *alarm event* that sets a timer. The action is defined by *activity* (see below).

CompensationHandler: This element allows a process to compensate if it fails to handle an error. Once a fault is signalled in a process, if the fault handler cannot handle the fault, the

compensation handler is invoked. Compensation handler performs a predefined action defined by *activity* (see below).

Activity: The BPEL specification provides a set of activities to specify the behaviour of a process. The *activity* in Figure 5.2 is a placeholder for any of these activities. In BPEL specification these activities are classified into two types: *basic* and *structured* activities. Here we briefly introduce all the activities provided by BPEL. A more detail description of each activity is provided in Sections 5.2.2.2 and 5.2.2.3.

```
<process name="ncname" targetNamespace="uri"
        queryLanguage="anyURI"?
        expressionLanguage="anyURI"?
        suppressJoinFailure="yes|no"?
        enableInstanceCompensation="yes|no"?
        abstractProcess="yes|no"?
        xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/">

<partners>?
     <partner name="ncname" serviceLinkType="qname"
          myRole="ncname"? partnerRole="ncname"?/>+
</partners>

<variables>?
     <variable name="ncname" messageType="qname"?
           type="qname"? element="qname"?/>+
</variables>

<correlationSets>?
    <correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>

<faultHandlers>?
     <catch faultName="qname"? faultContainer="ncname"?>*
        activity
     </catch>
     <catchAll>?
        activity
     </catchAll>
</faultHandlers>

<eventHandlers>?
     <onMessage partnerLink="ncname" portType="qname"
             operation="ncname" variable="ncname"?>
          activity
     </onMessage>
     <onAlarm for="duration-expr"? until="deadline-expr"?>*
         activity
     </onAlarm>
</eventHandlers>

<compensationHandler>?
         activity
</compensationHandler>

activity

</process>
```

Figure 5.2: The basic structure of BPEL business process [Bp03]

Basic activities in a BPEL process support primitive functions such as the invocation of operations and assignments of variable values. The basic activities in BPEL are:

(i) *invoke* activity – This activity invokes an operation in one of the partner services of the composition process.

(ii) *receive* activity – This activity makes the composition process to wait for the receipt of a message from a designated partner service.

(iii) *reply* activity – This activity makes the composition process to respond to a request previously accepted through a *receive* activity.

(iv) *assign* activity – This activity is used to assign the value of one variable to another variable and to initialise a variable with new data using value expressions.

(v) *throw* activity – This activity is used to signal an internal fault.

(vi) *wait* activity – This activity is used to specify a delay in the process that must last for a certain period of time.

(vii) *empty* activity – This activity is used to introduce a void operation that does nothing in a process.

The structured activities in BPEL provide the control and data flow structures that enable the composition of basic activities into a business process. These activities are:

(i) A sequence activity – This activity includes an ordered list of other activities that must be executed sequentially in the exact order of their listing.

(ii) A *switch* activity – This activity includes an ordered list of one or more conditional branches that include other activities. The conditional branches are considered in the order in which they are listed within a *switch* activity and the activity associated with the first branch whose condition is satisfied is executed. A *switch* activity may include a default branch that will be executed if none of the conditions of the other branches is satisfied.

(iii) A *flow* activity – This activity includes a set of two or more other activities that should be executed concurrently. A *flow* activity completes when all of the activities in it have

been completed. Synchronisation dependencies between activities inside a *flow* can be specified using *links*. Each link has a *source* and a *target* activity. The meaning of a link is that the target activity cannot start before the execution of the *source* activity of the link has been completed.

(iv) A *pick* activity – This activity contains an ordered list of one or more event and activity pair. This activity makes a composition process to wait for the occurrence of one of these events and then perform the activity associated with the event as soon as it occurs. A *pick* defines two types of events: (i) *message events* which signify the arrival of a message, and (ii) *alarm events* that set timers. Alarm event enable *pick* to execute a different activity if none of the expected message events happens within the time period specified by the timer.

(v) A *while* activity – This activity is used to specify the iterative execution of one or more activities as long some condition is valid.

5.2.2.2 Transformation of Basic BPEL Activities to EC

Invoke

The *invoke* activity allows an instance of a BPEL process to call an operation provided by a partner. This call can be asynchronous, i.e. an one-way call that does not expect any result from the called operation and in which the execution of the calling process can proceed, or synchronous, i.e. a request-response call in which the calling process expects a reply from the called operation and waits until it receives this reply. In case of an asynchronous one-way call, the *invoke* activity uses only a single input variable to send messages to the called operation. In case of a synchronous call, the *invoke* activity requires an additional output variable to deal with the response message produced by the called operation. An example of the *invoke* activity is shown below,

```
<invoke partner="Pname" portType="PTname" operation="Oname"
              inputVariable="IVname"? outputVariable="OVname"? />
```

This *invoke* activity, calls the operation *Oname* in the service that is bound to the partner *Pname* that deploys the port type *PTname* and *PTname* offers the operation, *IVname* and *OVname* are the input and output variables of the operation *Oname* respectively. This is an example of synchronous invoke as it involves an input variable and an output variable.

An asynchronous *invoke* activity calling an operation *O* is transformed into an atomic EC formula consisting of a Happens predicate that signifies the event of calling of *O*. For example, consider the following asynchronous *invoke* activity,

BPEL Activity	EC Representation
`<invoke partner="P" portType= "a:Pport" operation= "O" inputVariable = "X"/>`	**Happens**(in:P:O(_ID,_X.a),t1, ℜ(t1,t1))

In this example, the predicate **Happens**(in:P:O(_ID,_X.a),t1, ℜ(t1,t1)) signifies the occurrence of the event (i.e. in:P:O(_ID,_X.a)) of calling *O*. It should be noted that in the EC representation of this asynchronous *invoke*, the variable *_ID* takes as value a unique identifier that represents the exact instance of the BPEL process, and the variable *_X.a* takes the value of the part *a* of the input variable *X* of *O* at the time of the invocation[1].

A synchronous *invoke* activity calling an operation *O* is transformed into a conjunctive EC formula consisting of a *Happens* predicate that signifies the occurrence of the event denoting the call of *O*, a *Happens* predicate that signifies the occurrence of the event denoting the response from *O*, and a list of *Initiates* predicates, where each *Initiates* signifies the initiation of a fluent representing the value of a part of the output variable of *O*. For example, consider the following synchronous *invoke* activity,

BPEL Activity	EC Representation
`<invoke partner="P" portType= "a:Pport" operation= "O" inputVariable = "X" outputVariable= "Y"/>`	**Happens**(in:P:O(_ID,_X.a),t1, ℜ(t1,t1)) ∧ (∃t2) **Happens**(ir:P:O(_ID),t2, ℜ(t1,t2)) ∧ **Initiates**(ir:P:O(_ID),valueOf(Y.b,_Y.b),t2)

In this example, the predicate **Happens**(in:P:O(_ID,_X.a),t1, ℜ(t1,t1)) signifies the occurrence of the event (i.e. in:P:O(_ID,_X)) of calling *O*, the predicate **Happens**(ir:P:O(_ID),t2, ℜ(t1,t2)) signifies the occurrence of the event (i.e. ir:P:O(_ID)) of the response from *O*, and the predicate **Initiates**(ir:P:O(_ID),valueOf(Y.b,_Y.b),t2) signifies the initiation of a fluent (i.e. valueOf(Y.b,_Y.b)) representing the value of the part *b* of the variable *Y*. It should be noted that in the EC representation of this synchronous *invoke*, the variable *_ID* takes as value a unique identifier that represents the exact instance of the BPEL process, and the variable *_X.a* takes the value of the part *a* of the input variable *X* of *O* at the time of the invocation. Note that in this case, it is unrealistic to predict the precise time of the completion of the execution and return of *O* (due to the physical distribution of component services and/or network communication delays). Thus, the EC representation of *invoke* does not specify an upper limit for the time variable *t2* that signifies the time of the return of *O* and the range of *t2* is specified as *ℜ(t1,*

[1] For simplicity it is assumed that each BPEL variable used in the examples in Section 5.2.2.2 has single part

t2) in the EC representation. System providers may set an upper limit for this range depending on specific performance requirements that they may have.

Receive

A *receive* activity in BPEL specifies a message to be received from a designated partner and the content of the received message to be placed into a *variable*. A BPEL process exposes its operations by *receive* activities. This is done by mapping a WSDL operation onto the *receive* activity. The *receive* activity specifies the *partner* the process expects that will invoke the corresponding operation, and *portType* and *operation* that it expects the *partner* to invoke. The *receive* activity blocks the execution of all the activities that follow it until an appropriate message is received. The *receive* activity also enables to create a new process instance on receipt of a message. The basic syntax of the *receive* activity is shown below,

```
<receive partner="Pname" portType="PTname" operation="Oname"
                        variable="Vname"? createInstance="yes|no"? />
```

In the above *receive* activity *PTname* is the port type that offers the operation *Oname*. The process expects the partner *Pname* to invoke the operation *Oname*. *Vname* is the variable that holds the message received from *Pname*. The attribute *createInstance* set with value *yes*, allows the creation of a new instance of the process.

The transformation of a *receive* activity into EC requires a conjunction of a predicate that signifies the occurrence of the message receipt event and a list of predicates that initiate fluents representing the values of the received message. The pattern for transforming BPEL activities into an EC formula is given below.

BPEL Activity	EC Representation
`<receive partner="P" portType= "a:Pport" operation="O" variable="X"/>`	**Happens**(rc:P:O(_ID),t,ℜ(t,t)) ∧ **Initiates** (rc:P:O(_ID),valueOf(X.a,_X.a),t)

In this pattern, the predicate **Happens**(rc:P:O(_ID),t,ℜ(t,t)) signifies the occurrence of the message receipt event that invokes the operation O in a BPEL process (i.e. rc:P:O(_ID)) and the predicate **Initiates**(rc:P:O(_ID),valueOf(X.a,_X.a),t) signifies the initiation of a fluent (i.e. valueOf(X.a,_X.a)) representing the value of the part *a* of the variable *X*. The variable *_ID* takes as value a unique identifier that represents the exact instance of the message receipt event in the BPEL process instance.

Reply

The *reply* activity enables a BPEL process to send a response to a request received through a *receive* activity. The combination of *receive* and *reply* activities enables the specification of synchronous calls of BPEL process operations by web-services. A *reply* activity specifies the partner that made the request, the port type and the operation that the request was made for and the variable that holds the message data to be sent in reply. The basic syntax of the *reply* activity is shown below,

```
<reply partner="Pname" portType="PTname" operation="Oname"
                                                    variable="Vname"? />
```

In the above *reply* activity, *PTname* is the name of the port that offers the operation *Oname*, *Pname* is the partner that made the request for the operation, and *Vname* contains the message to be sent as a reply to the request.

The transformation of a *reply* activity into EC representation requires a single predicate that signifies the occurrence of the event denoting the reply to the request and takes place according to the following pattern.

BPEL Activity	EC Representation
`<reply partner="P" portType = "a:Pport" operation= "O" variable= "X"/>`	**Happens**(re:P:O(_ID, _X.a), t,ℜ(t,t))

In this pattern, the predicate **Happens**(re:P:O(_ID, _X.a), t,ℜ(t,t)) signifies the occurrence of the event that denotes the reply to the request (i.e. re:P:O(_ID, _X.a)) and the variable *_X.a* takes the value of the part *a* of the variable *X* at the time of reply. The variable *_ID* takes as value a unique identifier of the reply that allows the correlation (matching) of the particular instance of the *reply* activity with the corresponding instance of the *receive* activity that invoked the operation *O* in the specific instance of the BPEL process.

Assign

The *assign* activity in BPEL copies a value from a data element (source) to another data element (destination). The basic syntax of the *assign* activity is shown below

```
<assign>
      <copy>+
            from-spec
            to-spec
      </copy>
</assign>
```

BPEL defines several forms for specifying *from-spec* and *to-spec*. In all these forms, the type of the *from-spec* and the type of the *to-spec* must be compatible. The type compatibility of *from-spec* and *to-spec* in *assign* activity is described in [Bpe03]. In our framework we consider only the most commonly used forms of specifying *from-spec* and *to-spec*. These are:

```
<from variable="Vname" part="Pname"?/>
<from> ... literal value ... </from>
<to variable="Vname" part="Pname"?/>
```

In the above forms of *from-spec* and *to-spec*, *Vname* denotes a variable and *Pname* is a part of the variable *Vname*. Using the first combination of the *from-spec* and the *to-spec* data can be copied from one variable to another variable. Using the second combination of the *from-spec* and *to-spec* a variable can be initialised to a constant literal value.

Combining the *from-specs* with the *to-spec* we can have following two types of *assigns* in our framework,

Type-1	Type-2
<assign> <copy>+ <from variable="Vname" part="Pname"?/> <to variable="Vname" part="Pname"?/> </copy> </assign>	<assign> <copy>+ <from> ... literal value ... </from> <to variable="Vname" part="Pname"?/> </copy> </assign>

The transformation of a type-1 *assign* activity into EC requires a conjunction of a predicate that signifies the occurrence of the assignment event, a predicate that signifies the value of the part of the source variable and a predicate that initiates a fluent representing the assignment of this value to the part of the destination variable. The transformation of a type-2 *assign* activity into EC representation also requires a conjunction of a predicate that signifies the occurrence of the assignment event, a relational predicate that signifies the literal value of the source and a predicate that initiates a fluent representing the assignment of this value to the part of the destination variable. Consider the following two *assign* activities,

BPEL Activity	EC Representation
<assign name ="A"> <copy> <from variable ="X" part="a"/> <to variable="Y" part="b"/> </copy> </assign>	**Happens**(as:A(_ID), t1, ℜ(t1,t1)) ∧ **HoldsAt**(valueOf(x.a,_X.a),t1) ∧ (∃t2) (t1 < t2) ∧ **Initiates**(as:A(_ID), valueOf(Y.b, _X.a),t2)
<assign name ="B"> <copy> <from>zz</from> <to variable="Y" part="b"/> </copy> </assign>	**Happens**(as:B(_ID), t1, ℜ(t1,t1)) ∧ **Initiates**(as:B(_ID), valueOf(Y.b, _Y),t2) ∧ zz = _Y

In the first example (*assign A*), the predicate **Happens**(as:A(_ID), t1, ℜ(t1,t1)) signifies the occurrence of the assignment event (i.e. as:A(_ID)), the predicate **HoldsAt**(valueOf(x.a,_X.a),t1) signifies the value of the part *a* of the source variable *X* at the time of the execution of the assignment activity and the predicate **Initiates**(as:A(_ID), valueOf(Y.b,_X.a),t2) initiates a fluent (i.e. valueOf(Y.b,_X.a)) representing the assignment of this value to the part *b* of the destination variable *Y*. The variable *_ID* takes as value a unique identifier that represents the exact instance of this assignment event in the BPEL process instance.

In the second example (*assign B*), the predicate **Happens**(as:B(_ID), t1, ℜ(t1,t1)) signifies the occurrence of the assignment event (i.e. as:B(_ID)), the predicate **Initiates**(as:B(_ID), valueOf(Y.b,_Y),t2) initiates a fluent (i.e. valueOf(Y.b,_Y)) representing the assignment of this value to the part *b* of the destination variable *Y* and the relational predicate zz = _Y signifies the literal value that has been used to initiate the fluent. The variable *_ID* takes as value a unique identifier that represents the exact instance of this assignment event in the BPEL process instance.

Throw

The *throw* activity in BPEL enables a process to signal an internal fault explicitly. The *throw* activity specifies the name of the fault being thrown, and a variable that holds the data related to the fault which may be used by a fault handler to deal with the fault. The basic syntax of the *throw* activity is shown below,

```
<throw faultName="Fname" faultVariable="FVname"? />
```

In the above *throw* activity, *Fname* is the name of the fault thrown, and *FVname* is the variable that holds the fault data.

The transformation of a *throw* activity into EC representation requires a single predicate that signifies the occurrence of the fault event. Consider the following *throw* activity,

BPEL Activity	EC Representation
`<throw faultName="fN" faultVariable="X"/>`	**Happens**(th:fN(_ID, _X.a), t, ,ℜ(t,t))

In this example, the predicate **Happens**(th:fN(_ID, _X.a), t, ,ℜ(t,t)) signifies the occurrence of the fault event (i.e. th:fN(_ID, _X.a)), and the variable *_X.a* takes the value of the part *a* of the fault variable *X* at the time when the fault is thrown.

Wait

The *wait* activity allows suspending a process execution unconditionally. This suspension can be relative (i.e. for a fixed amount of time) or absolute (i.e. until certain date and time). The basic syntax of the *wait* activity is shown below,

```
<wait (for="duration-expr" | until="deadline-expr")/>
```

In the above *wait* activity, `duration-expr` and `deadline-expr` represents the duration or the deadline of the suspension respectively. Both the `duration-expr` and the `deadline-expr` are expressed using XPath *Expr* production [Xpa99]. In this transformation we consider only the `duration-expr`. This is because `deadline-expr` can be translated into `duration-expr` by taking difference between the current time and the deadline.

The transformation of a *wait* activity into EC requires a conjunction of predicates that signify the EC representation of the activities before and after the *wait* activity and a time predicate that compares the times of the activities before and after the *wait* activity. More specifically, a *wait* activity is transformed to EC according to the following pattern:

BPEL Activity	EC Representation

`<actType name="A">…</actType>`
`<wait for = "T"/>`
`<actType name="B">…</actType>`
	$\mathbf{EC}(A,[]) \wedge \mathbf{EC}(B,[]) \wedge$
	$\max_t(A) < \min_t(B) - T$

In the above example[2]

- *actType* can be any type of a basic or structured BPEL activity;
- $EC(X, [t_1,\ldots,t_n])$ denotes the EC formula (sub-formula) that activity X is transformed to;
- $min_t(X)$ represents the time of the earliest predicate in the EC representation of activity X, and $max_t(X)$ represents the time of the latest predicate in the EC representation of activity X.

The EC representation for the above BPEL example signifies that *activity B* (or the first *activity* in *activity B*, if *activity B* is a structured *activity*) starts *T* time units after the completion of *activity A* (or the last *activity* in *activity A*, if *activity A* is a structured activity).

[2] *acType*, $EC(X,[])$,$min_t(X)$ and $max_t(X)$ have the same semantic through out this chapter.

Empty

The activity *empty* in BPEL is used to introduce a void operation that does nothing in a process. A typical use of the *empty* activity is in cases where a specific fault is to be suppressed. The basic syntax of an *empty* activity is shown below,

```
<empty/>
```

empty activities are not represented in EC as the BPEL process itself does not perform any action in case of *empty* activities.

5.2.2.3 Transformation of BPEL Structured Activities to EC

Sequence

The *sequence* activity in BPEL specifies that a set of activities must be executed in a specific order in a process. The *sequence* activity encloses one or more activities that are to be executed in the order in which they appear in *sequence*. The basic syntax of the *sequence* activity is shown below,

```
<sequence >
      activity+
</sequence>
```

According to the above syntax, a *sequence* must contain at least one activity and this activity can be a basic activity or a structured activity.

The transformation of a *sequence* activity into EC requires a conjunction of predicates and time predicates, where each predicate in the conjunction represent the EC representation of the activities appear in the *sequence* and the time predicates represents the time relation between the predicates. Consider the following case,

BPEL Activity	EC Representation
<sequence> <actType name="A">…<actType> <actType name="B">…<actType> </sequence>	**EC**(A,[]) ∧ **EC**(B,[]) ∧ $max_t(A) < min_t(B)$

In the above example, the *sequence* activity has two activities A and B and the EC representation of the *sequence* signifies that *activity B* (or the first *activity* in *activity B*, if *B* is

a structured *activity*) starts only after the completion of *activity A* (or the last *activity* in *activity A*, if *activity A* is a structured activity).

Flow

Concurrent activities in BPEL are modelled using the *flow* activity. A *flow* activity specifies two or more sub-activities which are to be executed concurrently. The *flow* activity completes when each activity contained within the *flow* has completed. In some cases, the activities inside a *flow* require some degree of synchronization. BPEL offers the construct of *links* that can be used to establish synchronization dependencies between activities contained within a *flow*. Each *link* defines a *transition condition*, a *target* activity and a *source* activity. The *target* activity can start only when the *source* activity is completed and the *transition condition* is satisfied. If a transition condition is not defined, a *transition condition* with the default value *true* (i.e. a condition that is always satisfied) is assumed. In cases where dependencies between activities inside the *flow* are defined using *links*, at the beginning of the execution of the *flow*, only those *activities* with no dependencies (i.e., activities that are not defined as the *target* of any *link*) can be executed. The activities in the flow which constitute targets if links are executed only when their respective *source* activities are completed and the *transition conditions* of the relevant *links* are satisfied. The basic syntax of the *flow* activity is shown below,

```
<flow >
      <links>?
            <link name="Lname" transitionCondition="bool-expr"?/>+
      </links>
      activity+
</flow>
```

In the above *flow* activity, *Lname* is the name of a *link*, defined in the *flow*, *bool-expr* is the transition condition and *activity* can be any basic or structured activity. The transition condition *bool-expr* is expressed using XPath *Expr* production [Xpa99].

The transformation of a *flow* activity into EC representation requires a set of conjunctions of predicates, where each conjunction represents the EC representation of the synchronously dependent activities defined inside the *flow*. Consider the following case. In this example the *flow* activity encloses four activities, namely *A, B, C* and *D*. *Activity A* and *activity D* are independent of any other activities. *Activity B* is conditionally dependent on *activity A* and *activity C* is unconditionally dependent on *activity A*. Therefore this example contains 3 sequences of synchronously dependent (or independent) activities, these are *A-B, A-C* and *D*.

In the EC representation of the first sequence *activity B* (or the first *activity* in *activity B*, if *activity B* is a structured *activity*) starts only after the completion of *activity A* (or the last *activity* in *activity A*, if *activity A* is a structured activity) and the *transition condition* is satisfied after the completion of *activity A*, which is signified by the **HoldsAt**(valueOf(P,_p),t_2) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_2) ∧ _p =_v_1 predicates in the EC representation. The EC representation of the second sequence signifies that *activity C* (or the first *activity* in *activity C*, if *activity C* is a structured *activity*) starts only after the completion of *activity A* (or the last *activity* in *activity A*, if *activity A* is a structured activity). The EC representation of the third sequence signifies *activity D* is independent of any other activity.

BPEL Activity	EC Representation
<flow> <links> <link name="AtoB"/> <link name="AtoC"/> … </links> <actType name="A"> <source linkName="AtoB" transitionCondition="P=v_1"/> <source linkName="AtoC" /> … </actType> <actType name="B"> <target linkName="AtoB" /> … </actType> <actType name="C"> <target linkName="AtoC" /> … </actType> <actType name="D">… </actType> </flow>	**EC**(A,[]) ∧ **HoldsAt**(valueOf(P,_p),t_2) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_2) ∧ _p =_v_1 ∧ max_t(A) < t_2 ⇒ **EC**(B,[]) ∧ t_2 < min_t(B) **EC**(A,[]) ⇒ **EC**(C,[]) ∧ max_t(A) < min_t(C) **EC**(D,[])

Switch

The *switch* activity is used to specify conditional branching in a process. This activity includes an ordered list of one or more conditional branches where each branch can be a structured or a basic activity. The conditional branches are considered in order and the first branch whose condition is true is executed. In the case where the condition of no branch is true a default branch can be specified. The basic syntax of the *switch* activity is shown below,

```
<switch>
      <case condition="bool-expr">+
            activity
      </case>
      <otherwise>?
            activity
      </otherwise>
</switch>
```

In the above *switch* activity, *case* is the element that is used to define each conditional branch and *bool-expr* is the condition associated with each branch, where *bool-expr* is expressed using XPath *Expr* [Xpa99]. *Activity* can be any basic or structured activity, and *otherwise* is

the element that is used to define the default branch that will be executed if none of the branches has a satisfied condition.

The transformation of the *switch* activity into EC representation requires a set of conjunctions of predicates where each conjunction stands for the EC representation of each branch. Consider the following *switch* activity,

In this example, the *switch* activity contains two conditional branches and the default branch. In the first conditional branch, the *activity A* depends on the condition $P=v_1$. In the second conditional branch the *activity B* depends on the condition $P=v_2$ and the default branch contains the *activity C*. The EC representation for the first branch signifies that *activity A* (or the first *activity* in *activity A*, if *activity A* is a structured *activity*) starts only after the branch condition is satisfied at some time point (see the predicates **HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p =_v_1).

BPEL Activity	EC Representation
<switch> <case condition=" P=v_1"> <acType name="A"> … </acType> </case> <case condition=" P=v_2"> <acType name="B"> … </acType> </case> <otherwise> <acType name="C"> … </acType> </otherwise> </switch>	**HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p =_v_1 ⇒ **EC**(A,[]) ∧ t_1 < min_t(A) **HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p !=_v_1 **HoldsAt**(valueOf(v_2,_v_2),t_1) ∧ _p =_v_2 ⇒ **EC**(B,[]) ∧ t_1 < min_t(B) **HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p !=_v_1 **HoldsAt**(valueOf(v_2,_v_2),t_1) ∧ _p !=_v_2 ⇒ **EC**(C,[]) ∧ t_2 < min_t(C)

The EC representation for the second branch signifies that *activity B* (or the first *activity* in *activity B*, if *activity B* is a structured *activity*) starts only after the branch condition is satisfied and the first branch condition is not satisfied at some time point (see the predicates **HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p !=_v_1 **HoldsAt**(valueOf(v_2,_v_2),t_1) ∧ _p =_v_2). The EC representation of the default branch signifies that *activity C* (or the first *activity* in *activity C*, if *activity C* is a structured *activity*) starts only if none of the branch condition is satisfied at some time point (see predicates **HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p !=_v_1 **HoldsAt**(valueOf(v_2,_v_2),t_1) ∧ _p !=_v_2).

While

The *while* activity enables iterative execution of activities as long as a condition is satisfied. The basic syntax of the *while* activity is shown below,

```
<while condition="bool-expr">
      activity
</while>
```

In this *while* activity, *activity* will be executed as long as the condition *bool-expr* is satisfied. *activity* can be a basic or a structured activity. The condition *bool-expr* is expressed using XPath *Expr* production [Xpa99].

A *while* activity is transformed into EC using the following pattern,

BPEL Activity	EC Representation
<while condition="P=v_1"> <acType name="A"> … </acType> </while>	**HoldsAt**(valueOf(P,_p),t_1) ∧ **HoldsAt**(valueOf(v_1,_v_1),t_1) ∧ _p =_v_1 ⇒ **EC**(A,[]) ∧ t_1 < min_t(A)

According to this pattern, the activity *A* will be executed as long as the condition $p=v_1$ is satisfied.

Pick

The *pick* activity awaits the occurrence of an event that belongs to a pre-defined set of events and performs the activity associated with the specific type of event that has occurred. The *pick* activity is similar to the *switch* activity except that the selection of the activity to be executed in *pick* depends on the occurrence of an event rather than the evaluation of a condition. Two types of events can be defined in *pick* activity, (i) *message events* which signify the arrival of a message, and (ii) *alarm events* based on a timer. An alarm event enables pick to execute a different activity in cases where none of the expected message events happens within the time period specified by the timer. The *pick* activity also allows to create a new process instance on the arrival of a message i.e. on the occurrence of a *message event*. The basic syntax of the *pick* activity is shown below,

```
<pick createInstance="yes|no"?>
      <onMessage partner="Pname" portType="PTname"
            operation="Oname" variable="Vname"?>+
                  activity
      </onMessage>
      <onAlarm (for="duration-expr" | until="deadline-expr")>*
            activity
      </onAlarm>
</pick>
```

In the above *pick* activity, the message events are defined by the element *onMessage*. More specifically, *PTname* is the port type that offers the operation *Oname*, *Pname* is the partner that is to invoke the operation *Oname* and *Vname* is the variable that holds the message

received from *Pname*. The activity to be performed on occurrence of the message event is *activity*. The element *onAlarm* defines the alarm events. *duration-expr* and *deadline-expr* represent the duration or the deadline of the timer. Both the *duration-expr* and the *deadline-expr* are expressed using XPath *Expr* production [Xpa99]. The transformation of *pick* into EC takes into account only the *duration-expr*, since `deadline-expr` can be translated into `duration-expr` by taking the difference between the current time and the deadline. The attribute *createInstance* of the *pick* activity determines whether or not a new instance of the process will be created (a new instance will be created if the value of the attribute is *yes*) on the arrival of new message.

The transformation of a *pick* activity into EC is based on the following pattern:

BPEL Activity	EC Representation
<sequence> <acType name="A"> … </acType> <pick> <onMessage partner="P" portType="a:Pport" operation="O" variable="X"> <acType name="B"> … </acType> </onMessage> <onAlarm for="T"> <acType name="C"> … </acType> </onAlarm> </pick> </sequence>	**EC**(A,[]) ∧ **Happens**(rc:O(_ID), t_2,ℜ(max_t(A), max_t(A) + T)) ∧ **Initiates**(rc:O(_ID), valueOf(X,_X.a),t_2) ⇒ **EC**(B,[]) ∧ t_2 < min_t(B) **EC**(A,[]) ∧ ¬ **Happens**(rc:O(_ID), t_2,ℜ(max_t(A), max_t(A) + T)) ⇒ **EC**(C,[]) ∧ max_t(A) + T < min_t(C)

In the above pattern, the *pick* activity contains one message event and one alarm event. After the completion of the activity *A* the process waits *T* time units for the message event. If the message event occurs within *T* time units, the activity *B* is executed. If the message event does not occur within *T* time units after the completion of A, the activity *C* is executed. The EC representation for the message event branch signifies that the message event may occur within *T* time units after the occurrence of the activity *A* (see the predicate **Happens**(rc:O(_ID), t_2,ℜ(max_t(A), max_t(A) + T))) and the activity *B* (or the first *activity* in *activity B*, if *activity B* is a structured *activity*) starts only after the occurrence of the message event. The EC representation of the event branch signifies that the activity *C* (or the first *activity* in *activity C*, if *activity C* is a structured *activity*) starts only *T* time units after the completion of the activity *A*, provided that the message event does not occur within *T* time units after the completion of activity *A* (see the predicate ¬ **Happens**(rc:O(_ID), t_2,ℜ(max_t(A), max_t(A) + T))).

5.2.2.4 Transformation of Miscellaneous BPEL Activities to EC

FaultHandlers

The *faultHandlers* activity in BPEL enables a process to handle errors found within the process. This activity contains a list of fault and activity pair. If a fault in the list occurs the associated activity is performed. The basic syntax of the *faultHandlers* activity is shown below,

```
<faultHandlers>?
      <catch faultName="Fname"? faultVariable="FVname"?>*
            activity
      </catch>
      <catchAll>?
            activity
      </catchAll>
</faultHandlers>
```

In a *faultHandlers* activity, a *catch* element is used to handle a specific fault within the process, *Fname* is the name of the fault, *Fvname* is the variable that holds fault related information and *activity* is the activity to be performed if the fault occurs. The *catchAll* element inside the *faultHandlers* is used to handle any fault not handled by a more specific *catch* element by performing the *activity* inside the *catchAll* element.

The transformation of the *faultHandlers* into EC requires a set of conjunctions of predicates where each conjunction stands for EC representation of each *catch* and *catchAll* element. Consider the following *faultHandlers*

BPEL Activity	EC Representation
<faultHandlers> <catch faultName="F" faultVariable="Y"> <acType name="A"> … </acType> </catch> <catchAll> <acType name="B"> … </acType> </catchAll> </faultHandlers>	**Happens**(th:F(_ID), t_1,$\Re$(t_1, t_1)) ∧ **Initiates**(th:F(_ID),valueOf(Y.a,_Y.a),t_1) ⇒ **EC**(A,[]) ∧ $t_1 < min_t$(A) **Happens**(th:XX(_ID), t_1,$\Re$(t_1, t_1)) ∧ ¬**Happens**(th:F(_ID), t_1,$\Re$(t_1, t_1)) ⇒ **EC**(B,[]) ∧ $t_1 < min_t$(B)

In the above example, the activity *A* is performed if the fault *F* is thrown and the activity *B* is performed if a fault other than *F* is thrown. The EC representation for the *catch* element signifies that the activity *A* (or the first *activity* in *activity A*, if *activity A* is a structured *activity*) starts only after the fault *F* occurs (see the predicate **Happens**(th:F(_ID), t_1,$\Re$(t_1, t_1))). Since the *catchAll* element is independent of any specific fault other than the fault *F* its EC representation signifies that the activity *B* (or the first *activity* in *activity B*, if *activity B* is a structured *activity*) starts only after a fault occurs, other than the fault *F* (see the predicates **Happens**(th:XX(_ID), t_1,$\Re$(t_1, t_1)) ∧ ¬**Happens**(th:F(_ID), t_1,$\Re$(t_1, t_1)) here XX denotes any fault).

EventHandlers

An *eventHandlers* activity allows a BPEL process to handle events asynchronously. This activity contains a list of event and activity pairs. If an event in the list occurs, the associated activity is performed. Two types of events can be defined in *eventHandlers*,

(i) *message event* which signifies the arrival of a message, and

(ii) *alarm event* based on a timer.

The basic syntax of an *eventHandlers* activity is shown below,

```
<eventHandlers>?
     <onMessage partnerLink="ncname" portType="qname"
             operation="ncname" variable="ncname"?>
           activity
     </onMessage>
     <onAlarm for="duration-expr"? until="deadline-expr"?>*
         activity
     </onAlarm>
</eventHandlers>
```

In the above *eventHandlers* activity, the message event is defined by the element *onMessage*. *PTname* is the port type that holds the operation *Oname*. The process expects the partner *Pname* to invoke the operation *Oname*. *Vname* is the variable that holds the message received from *Pname*. *Activity* is the activity to be performed on occurrence of the message event. The element *onAlarm* defines the alarm events. `duration-expr` and `deadline-expr` represents the duration or the deadline of the timer. Both the `duration-expr` and the `deadline-expr` are expressed using XPath *Expr* [Xpa99]. In this transformation we consider only the `duration-expr`, since `deadline-expr` can be translated into `duration-expr` by taking difference between the current time and the deadline. The syntax of the *eventHandlers* activity is similar to that of the *pick* activity and the EC representation of *eventHandlers* is same as that of the *pick* activity as defined in Section 5.2.2.3.

5.2.2.5 Extraction of Formulas from BPEL Specification

In this section, we describe the mechanism used to extract formulas from BPEL process specification with the help of an illustrative example. The steps followed to extract formulas from BPEL specification are as follows,

- Identify all the execution paths (i.e. logical sequences of BPEL activities) in a BPEL process specification

- For each sequence of BPEL activities ($Seq\text{-}BPEL_i$) transform the BPEL activities into EC representation by applying the transformation patterns discussed in Sections 5.2.2.2, 5.2.2.3 and 5.2.2.4 and produce a sequence of EC predicates $Seq\text{-}EC_i$.

- Transform each sequence of EC predicates ($Seq\text{-}EC_i$) into an EC formula by identifying the head and the body of the formula. The identification of the head and the body mainly depends on the position of the implication sign in the sequence of the EC predicates. The implication sign in $Seq\text{-}EC_i$ is introduced by the transformation of BPEL structured activities into EC representation. Because of nested structured activities more than implication sign can be introduced in an EC predicate sequence. To identify the head and the body of formula, i.e. the appropriate position of the implication sign, we apply the patterns shown below. These patterns are expressed in some pseudo language.

Pattern -1	**Pattern -2**	**Pattern -3**

{
statement-1
statement-2
statement-3
}
	{
	statement-1
	⇒
	statement-2
	statement-3
	}
		{
		statement-1
		⇒
		statement -2
		⇒
		statement-3
		}

Statement-1 ∧ statement-2 ⇒ statement-3	Statement-1 ∧ condition ⇒ Statement-2 ∧ statement-3	Statement-1 ∧ condition-1 ∧ statement-2 ∧ condition-2 ⇒ statement-3

Pattern-1 specifies that if there is a sequence of statements that does not contain any implication sign, then the last statement should form the head of the formula. All the statements other than the last statement should form the body of the formula and an implication sign should be introduced in between the head and the body. This is because, the last statement will be executed only if all the statements precede it, execute successfully, i.e. execution of *statement-1* and *statement-2* implies the execution of *statement-3*. According to this pattern, if there is no implication sign in $Seq\text{-}EC_i$, then all the predicates in $Seq\text{-}EC_i$ that correspond to the last BPEL activity in $Seq\text{-}BPEL_i$ form the head of the formula and all other predicates form the body of the formula.

Pattern-2 says if there is a sequence that has one implication sign and one or more statements, then all the statements that appear after the implication sign should form the head of the formula. All the statements that appear before the implication sign should form the body of the formula. This is because, the execution of the statements that appear after the implication sign is only possible if all the statements appear before the implication sign execute successfully. According to this pattern, if there is an implication

sign in *Seq-EC*$_i$, then all the predicates after the implication form the head of the formula and all the predicates that appear before the implication sign form the body of the formula.

Pattern-3 says that if there is a sequence that has more than one implication sign and one or more statements, then all the statements that appear after the last implication sign should form the head of the formula. All the statements that appear before the last implication sign should form the body of the formula. Each implication sign except the last implication sign is replaced by a conjunctive logical operator ($\wedge$). This is because, the execution of the statements that appear after the last implication is only possible if all the statements appear before the last implication execute successfully. According to this pattern, if there are more than one implication sign (this can result from nested BPEL structured activities) in *Seq-EC*$_i$, then

- each implication sign except the last implication sign is replaced by a conjunctive logical operator ($\wedge$),
- all the predicates that appear after the last implication form the head of the formula and;
- all the predicates that appear before the last implication sign form the body of the formula.

Formula Extraction Example

In the following, we illustrate the formula extraction mechanism described above with an example. In our example, we use the *RateTrackerProcess* BPEL process shown in Figure 5.3.

This process allows users to convert any amount from one country currency to another country currency. Initially, the process receives the amount from customer (see *getAmount* receive activity in Figure 5.3). If the received amount is negative the process replies to the customer with 0. If the received amount is not negative the process waits for 30 seconds to receive the names of two countries from the customer (see *getConversion* pick activity in Figure 5.3). If the process receives two country names from the customer within 30 seconds, it calls the operation *getRateRequest* in the web service *currencyExchanger* to convert the received amount from first country currency to the second country currency (see *requestRate* invoke activity in Figure 5.3). Eventually, it replies to the customer with the converted amount (see *reply-amount* reply activity in Figure 5.3). If the process does not receive the two country names within 30 seconds, it replies to the customer with the received amount.

```
<process name="rateTrackerProcess"
  targetNamespace="http://tempuri.org/services/PriceTrackerBpel" …. >
  <partners> …..  </partners>
  <variables> … … </variables>
  <sequence>
    <receive name="getAmount" partner="customer" portType="wns:RateTrackerPortT"
                operation="getAmount" variable="amountRequest" createInstance="yes"/>
    <switch name="checkAmount">
      <case name="amountNegative" condition="bpws:getVariableData('amountRequest',
                                                                  'amount') < 0">
        <sequence>
          <assign name="assign1"> <copy>
              <from expression="0"/>
              <to variable="amountResponse" part="amount"/>
            </copy> </assign>
          <reply name="negative-reply" partner="customer"
                    portType="wns:RateTrackerPortT" operation="getAmount"
                                                        variable="amountResponse"/>
        </sequence>
      </case>
      <case name="amountPositive" condition="bpws:getVariableData('amountRequest',
                                                                  'amount') > 0">
        <pick name="getConversion">
          <onMessage name="conversion" partner="customer"
                    portType="wsdlns:RateTrackerPortType" operation="getConversion"
                                                    variable="conversionRequest">
            <sequence>
              <assign name="assign2"> <copy>
                   <from variable="conversionRequest" part="country1"/>
                   <to variable="rateRequest" part="currency1"/>
                 </copy>
                 <copy>
                    <from variable="conversionRequest" part="country2"/>
                    <to variable="rateRequest" part="currency2"/>
                 </copy>
                 <copy>
                    <from variable="amountRequest" part="amount"/>
                    <to variable="rateRequest" part="amount"/>
                 </copy> </assign>
              <invoke name="requestRate" partner="currencyExchanger"
                 portType="xns:GetCurrencyExchangeSOAP" operation="getRateRequest"
                   inputVariable="rateRequest" outputVariable="rateResponse"/>
              <assign name="assign3"> <copy>
                      <from variable="rateResponse" part="value"/>
                      <to variable="conversionResponse" part="amount"/>
                   </copy> </assign>
              <reply name="reply-Conversion" partner="customer"
                          portType="wns:RateTrackerPortT" operation="getConversion"
                                                    variable="conversionResponse"/>
            </sequence>
          </onMessage>
          <onAlarm name="noConversion" for="'PT30S'">
              <sequence>
                <assign name="assign4"> <copy>
                      <from variable="amountRequest" part="amount"/>
                      <to variable="amountResponse" part="amount"/>
                   </copy> </assign>
                <reply name="reply-amount" partner="customer"
                       portType="wns:RateTrackerPortT" operation="getAmount"
                                                      variable="amountResponse"/>
              </sequence>
            </onAlarm>
          </pick>
        </case>
      </switch>
    </sequence>
</process>
```

Figure 5.3: Rate tracker BPEL process

The complete specification of the *RateTrackerProcess* (i.e. the BPEL file and the WSDL files) is provided in Appendix F.

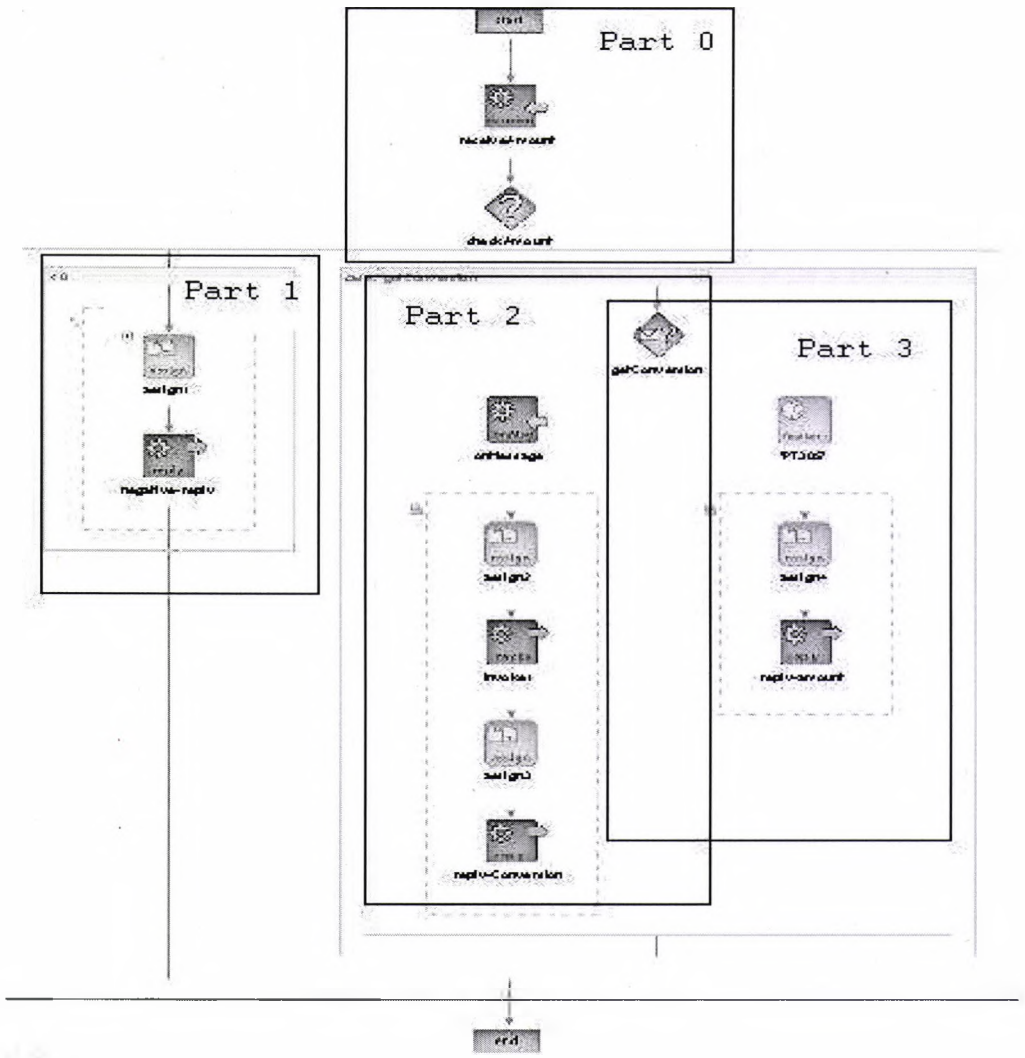


Figure 5.4: Sequences of activities in the *Rate Tracker* BPEL process

The execution paths (i.e. sequences of activities) of the *RateTrackerProcess* are shown in the Figure 5.4. The *RateTrackerProcess* has three execution paths:

Part 0 -Part 1: This sequence includes the *getAmount* receive activity and the first *case* (named as *amountNegative)* of the *switch* activity. Inside the *amountNegative* case there is a *sequence* activity which contains an assign activity (named as assign1) and a reply (named as *negative-reply)* activity.

Part 0 -Part 2: This sequence includes the *getAmount* receive activity, part of the second case (named as *amountPositive*) in the *switch* activity. The *amountPositive* case contains a pick (named as *getConversion*) activity and this sequence considers the *onMessage* (named as *conversion*) activity of the pick activity. The *conversion* onMessage activity contains a seqeucne activity and inside the secquence there is one assign activity (named as *assign1*), one invoke activity (namded as *requestrate*) and one reply activity (named as *reply-conversion*).

```
(B1) forall t1 : time, exists t2, t3, t4, t5 : time
Happens(rc:getAmount(ID),t1,R(t1,t1)) ^
Initiates(rc:getAmount(ID),valueOf(amount,vamount),t1) ^
HoldsAt(valueOf(amount,vamount),t2) ^ vamount < 0 ^ t1 <= t2
==>
Happens(as:assign1(ID),t3,R(t2,t3)) ^
Initiates(as:assign1(ID),valueOf(amount,vamount),t4) ^ t3 <= t4 ^
Happens(re:getAmount(ID,amount),t5,R(t4,t5))

(B2) forall t1 : time,
     exists t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12 : time
Happens(rc:getAmount(ID),t1,R(t1,t1)) ^
Initiates(rc:getAmount(ID),valueOf(amount,vamount),t1) ^
HoldsAt(valueOf(amount,vamount),t2) ^ vamount > 0 ^ t1 <= t2 ^
Happens(rc:getConversion(ID),t3,R(t2,t2+30000)) ^
Initiates(rc:getConversion(ID),valueOf(country2,vcountry2),t3) ^
Initiates(rc:getConversion(ID),valueOf(country1,vcountry1),t3)
==>
Happens(as:assign2(ID),t4,R(t3,t4)) ^
Initiates(as:assign2(ID),valueOf(currency1,vcurrency1),t5)^ t4<=t5 ^
Initiates(as:assign2(ID),valueOf(currency2,vcurrency2),t6)^ t5<=t6 ^
Initiates(as:assign2(ID),valueOf(amount,vamount),t7) ^ t6 <= t7 ^
Happens(in:getRateRequest(ID,currency2,currency1,number),t8,R(t7,t8))
^
Happens(ir:getRateRequest(ID),t9,R(t8,t9)) ^
Initiates(ir:getRateRequest(ID),valueOf(value,vvalue),t9) ^
Happens(as:assign3(ID),t10,R(t9,t10)) ^
Initiates(as:assign3(ID),valueOf(amount,vamount),t11) ^ t10 <= t11 ^
Happens(re:getConversion(ID,result),t12,R(t11,t12))

(B3) forall t1 : time, exists t2, t3, t4, t5, t6 : time
Happens(rc:getAmount(ID),t1,R(t1,t1)) ^
Initiates(rc:getAmount(ID),valueOf(amount,vamount),t1) ^
HoldsAt(valueOf(amount,vamount),t2) ^ vamount > 0 ^ t1 <= t2 ^
¬ Happens(rc:getConversion(ID),t3,R(t2,t2+30000))
==>
Happens(as:assign4(ID),t4,R(t3,t4)) ^
Initiates(as:assign4(ID),valueOf(amount,vamount),t5) ^ t4 <= t5 ^
Happens(re:getAmount(ID,amount),t6,R(t5,t6))
```

Figure 5.5: Formulas extracted for the Rate Tracker BPEL process

Part 0 -Part 3: This sequence includes the *getAmount* receive activity, part of the second case (named as *amountPositive*) in the *switch* activity. The *amountPositive* case contains a pick (named as *getConversion*) activity and this sequence considers the *onAlarm* (named as

noConversion) activity of the pick activity. The *noConversion* onAlarm activity contains a seqeucne activity and inside the secquence there is one assign activity (named as *assign3*), and one reply activity (named as *reply-amount*).

By applying the BPEL–EC transformation patterns discussed in Sections 5.2.2.2 – 5.2.2.5 and we will get the formulas for the *RateTrackerProcess* BPEL process which are shown in Figure 5.5.

5.2.2.6 IBehaviouralPropertiesExtractor

The interface *IBehaviouralPropertiesExtractor* consists of the following methods,

- `xsd:string extractFormulas(BPELFile xsd:string, WSDLFiles[] xsd:string)` – This method is used to extract behavioural properties from a BPEL specification. This method accepts the BPEL file name that specifies the BPEL process and the name of the WSDL file for each web service which is used by the process and returns the string representation of an XML document [Xml04c]. The returned string contains the XML representation of all the event calculus formulas extracted from the BPEL process. The XML document is written according to the schema discussed in Section 3.3.5 in Chapter 3.

- `void saveFormulas(formulas xsd:string, file xsd:string)` – This method is used to save formulas in an XML file. The parameter *formulas* is the string representation of an XML document that contains the formulas to be saved and the parameter *file* specifies the XML file.

- `xsd:string importFormulas(file xsd:string)` – This method is used to import formulas from an XML file. The parameter *file* refers the XML file which the formulas will be imported from. The method returns the string representation of an XML document that contains all the formulas imported from the XML file.

5.2.3 Event Receiver

During the execution of the composition process, the service based system execution environment writes the log events as strings to the event port specified in the monitoring

policy that has been given to the monitoring framework. The *event receiver* reads the string streams from the event port and identifies the type of the event that the string stream describes, filters out events, which are irrelevant to the monitoring process. The events of the process execution engine, which are irrelevant, are determined by event patterns. These patterns are identified by the *monitor manager* from the formulas that have been set for monitoring. In this section we discuss the mechanism used to realise the functionality of the *event receiver*. The *event receiver* offers its functionality through the interface *IEventReceiver*. We also describe the interface *IEventReceiver* at the end of the section.

The exact mechanism used to generate log events depends on the web service execution engine that is used. In our framework we have used the engine *bpws4j* [Bpw03] that uses log4j [Log03] to generate log events. Figure 5.6 shows the schematic diagram to generate log events in our framework. The system provider sets some log4j properties of the *bpws4j* engine to specify the level of event reporting (INFO, DEBUG etc.), and the destination of the logged events.

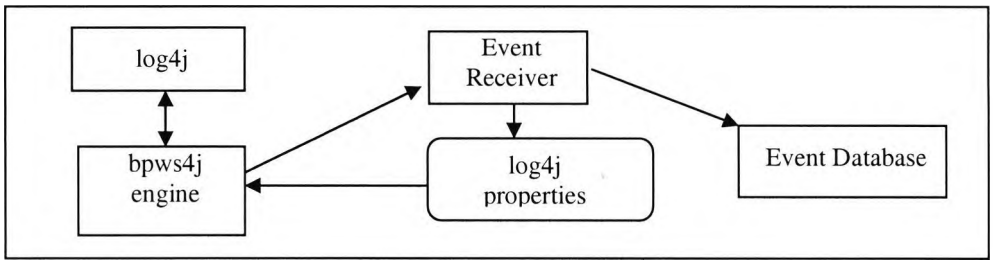


Figure 5.6: Event generation from *bpws4j* engine

The destination where the log events should be reported to is specified by: (i) the IP address of the host on which event receiver is running and (ii) the port number to which the *event receiver* is listening.

At runtime, the *bpws4j* engine generates log events according to the properties of log4j that have been set by the system provider and reports them to the destination. The *event receiver* (which is implemented as a remote *log4j* server) captures the log events which are reported to the port that it listens to and converts them to monitoring events. The transformation of log events into monitoring events is performed on parsing the former events by applying regular expressions in order to identify the required information from log events. In the following section, we describe this transformation.

5.2.3.1 Transformation of log events into Monitoring events

In this transformation we only use a sub set of the regular expression constructs defined in [Reg02]. The constructs used in the transformation are,

- Range construct "`-`", that signifies a range. For example `c-g` denotes any letter from `c` to `g` inclusive.
- Character Class "[ ]", that signifies character classes. For example, `[a-z0-9]` denotes any letter `a` through `z` or any digit `0` through `9`.
- Modifier "*", that signifies zero or more times repetition of the previous expression. For example `[ab]*` denotes `a, ab, aab` and so on

Any other character is considered as ordinary character. Regular expressions are expressed inside double quotation.

In addition to the above regular expression constructs the following definitions are used to express the transformations

- *LogString* is the string representation of the log event received by the *event receiver* from the *bpws4j* engine.
- *SubString($sting, $substring)* is a function that extracts a substring at the beginning of *$string* that matches *$substring* where *$substring* is a regular expression.
- *Matches($string, $substring)* is a function that returns *true* if *$substring* appears in *$string* and *false* otherwise. *$substring* is a regular expression.
- *Concat($string1, $string2)* is a function that performs the concatenation of *$string1* and *$string2* and returns the concatenated string. *$string2* can be a regular expression.
- *Subtract($string1, $string2)* is a function that removes the first occurrence of *$string2* from *$string1* and returns the new *$string1*. *$string2* can be a regular expression.
- *$string1= $string2* the value of *$string2* is assigned to *$string1*.
- φ denotes an empty string.

Generic Structure of Log events

The log events generated by *bpws4j* engine for different BPEL activities have a common generic structure. In this section we describe the generic structure of log events and the

regular expressions that are used to capture information from run time events that are applicable to any BPEL activity. The structure of log event (i.e. *LogString*) is shown below,

Event_Time_Stamp Event_ID Report_Level Event_Description

As shown above a log event has four columns. These are

Event_Time_Stamp signifies the occurrence time of the event, which appears in ISO 8601 format [Iso04]. This format is `YYYY-MM-DD hh:mm:ss,s` where,

YYYY = four-digit year
MM = two-digit month (01=January, etc.)
DD = two-digit day of month (01 through 31)
hh = two digits of hour (00 through 23)
mm = two digits of minute (00 through 59)
ss = two digits of second (00 through 59)
s = one or more digits representing a decimal fraction of a second

An example of *Event_Time_Stamp* is `2005-05-24 17:02:03,510`. The event receiver converts the time stamp into milliseconds, which is the number of milliseconds elapsed from 1/1/1970 until log event was created. This is done by taking the difference between the *Event_Time_Stamp* and `1970-01-01 00:00:00,000`.

Event_ID signifies the unique id assigned to this log event by the *bpws4j* engine. The format of the *EVENT_ID* is `http<number>-Processor<number>` or `Thread-<number>`. An example of *Event_ID* is `http8080-Processor21` or `Thread-22`. The event receiver applies the following regular expressions to extract the *Event_ID*,

```
SubString($LogString, "http[0-9][0-9][0-9][0-9]-Processor[0-9]*")
```

```
SubString($LogString, "Thread-[0-9]*")
```

Report_Level signifies the level of event reporting (e.g. INFO, DEBUG etc.) This information is not used by the event receiver.

Event_Description contains BPEL activity specific information as well as some generic information. Among the generic information it contains,

- The ID of the service (partner ID) appears as a sub string in the *Event_Description*, which takes the form `serviceID=<string>` where `<string>` represents the service ID. An example of a sub string that contains service ID is

```
serviceID='{http://carservice.org/wsdl/Online}carServiceBP'
```

The event receiver applies the following regular expression to extract the sub string that contains the service ID,

```
SubString($LogString,"serviceID=" + "'[a-zA-Z0-9_£$:}{/-]*'")
```

- The name of the operation associated with a BPEL activity appears as a sub string in the *Event_Description*, which takes the form `operationName=<string>` where `<string>` represents the operation name. An example of a sub string that contains operation name is,

```
operationName = 'receiveRequest'
```

The event receiver applies the following regular expression to extract the sub string that contains the operation name,

```
SubString($LogString,"operationName="+"'[a-zA-Z0-9_£$:}{/-
]*'")
```

- The names of the variables, their values and types appear as sub strings in the *Event_Description*, where each sub string take the form `JROM<type>:<string1>: <string2>` where `<type>` represents the variable type (e.g. string, int etc), `<string1>` represents the variable name, and `<string2>` represents the value of the variable. An example of a sub string that contains the variable description is,

```
JROMString: custId: kmr
```

The regular expression applied by the event receiver to extract the sub string that contains variable name, type and value is shown below. Since multiple variables can be associated with one operation, the regular expression should be applied multiple times to extract each variable. We show this with a while loop in a pseudo language,

```
$varString= φ
while(Matches($LogString, "JROM[a-zA-Z0-9_£$: ]*")
begin
   $varString=SubString($LogString, "JROM[a-zA-Z0-9_£$: ]*")
   $LogString=Subtract($LogString, $VarString)
end
```

Generation of Monitoring Events for Receive Activities

An example of a log event that represents an instance of the execution of a *receive* activity is shown below,

```
2005-05-24 17:02:03,510  [http8080-Processor21] INFO
     bpws.runtime - Incoming request:[WSIFRequest: serviceID =
     '{http://carservice.org/wsdl/OnlineRenter}carServiceBP'
     operationName = 'receiveRequest'
     incomingMessage = 'org.apache.wsif.base.WSIFDefaultMessage@70ccb name:null
     parts[0]:[JROMString: loc: One]  parts[1]:[JROMString: custId: kmr]'
     contextMessage = 'org.apache.wsif.base.WSIFDefaultMessage@d9230a name:null
     parts[0]:http://xml.apache.org/soap/v1
     parts[1]:{http://carservice.org/wsdl/OnlineRenter}CarRenter  parts[2]:CRS']
```

As shown in this example, the fact that this log event denotes a *receive* activity is indicated by the sub-string "`Incoming request`" in the *Event_Description*. The event receiver identifies this log event as a *receive* activity if the following condition holds,

```
Matches($LogString, "Incoming request")
```

The time stamp, event ID, service ID, operation name and variables are extracted by applying the regular expressions discussed for the generic structure of log events above.

Following the extraction of the above information, the event receiver is ready to generate monitoring events. As discussed in Section 5.2.2.2 a *receive* activity is transformed into a conjunction of EC predicates that signify the occurrence of the message receipt event and initiation of each variable due to the received message, the event receiver generates the following three monitoring events from this instance of log event

```
Happens(rc:{http://carservice.org/wsdl/OnlineRenter}carServiceBP:receiveRequest(http80
80-Processor21), 1116950523510, ℜ(1116950523510, 1116950523510))
Initiates(rc:{http://carservice.org/wsdl/OnlineRenter}carServiceBP:receiveRequest([htt
p8080-Processor21], valueOf(loc, _loc), 1116950523510)
Initiates(rc:{http://carservice.org/wsdl/OnlineRenter}carServiceBP:receiveRequest([htt
p8080-Processor21], valueOf(custId, _custId), 1116950523510)
```

Generation of Monitoring Events for Reply Activities

An example of a log event that signifies a *reply* activity is shown below,

```
2005-05-24 17:02:08,797  [http8080-Processor21] INFO
       bpws.runtime - Outgoing response: [WSIFResponse: serviceID =
       '{http://carservice.org/wsdl/OnlineRenter}carServiceBP'
       operationName = 'receiveRequest'
       isFault = 'false' outgoingMessage =
       'org.apache.wsif.base.WSIFDefaultMessage@b10bd5 name:null
       parts[0]:[JROMString: carId: k9h]' faultMessage = 'null'
       contextMessage = 'null']
```

The reply activity represented by the above log event corresponds to the log event of the *receive* activity presented earlier in this section.

The fact that the above log event denotes a *reply* activity is indicated by the sub string '`Outgoing response`' in the *Event_Description*. The *event receiver* identifies this log event as a *reply* activity if the following condition holds,

```
Matches($LogString, "Outgoing response")
```

The regular expressions for extracting the time stamp, partner ID, operation name and variables are the same as the regular expressions presented for the generic structure of log events above. Following the extraction of the above types of information, the event receiver generates the following monitoring event from this instance of log event,

```
Happens(re:{http://carservice.org/wsdl/OnlineRenter}carServiceBP:receiveRequest(http80
80-Processor21, carId), 1116950528797, ℜ(1116950528797, 1116950528797))
```

Generation of Monitoring Events for Invoke Activities

An example of a log event that signifies the invocation of an external service is shown below,

```
2005-05-24 17:02:03,680  [Thread-35] DEBUG
       bpws.runtime.bus - Invoking external service with [WSIFRequest: serviceID =
       '{http://tempuri.org/services/CustomerReg}CustomerRegService15d8362-
       1040f701b25--8000' operationName = 'authenticate'
       incomingMessage = 'org.apache.wsif.base.WSIFDefaultMessage@1667f3c name:null
       parts[0]:[JROMString: custId: kmr]'
       contextMessage = 'null']
```

The fact that the above log event denotes an invoke activity that calls an external service is indicated by the sub string '`Invoking external service`'. The *event receiver* identifies this log event as an invocation to external service if the following condition holds,

```
Matches($LogString, "Invoking external service")
```

The regular expressions for extracting the time stamp, event ID, partner ID, operation name and variables are the same as the regular expressions presented for the generic structure of log events above. Following the extraction of the above types of information, the event receiver generates the following monitoring event from this instance of the log event,

```
Happens(in:{http://tempuri.org/services/CustomerReg}CustomerRegService15d8362-1040f70
1b25--8000:authenticate(Thread-35, custId), 1116950523680, ℜ(1116950523680,
1116950523680))
```

The *log event* that signifies the response from the external service which corresponds to the invocation represented by the above log event is shown below,

```
2005-05-24 17:02:08,236  [Thread-35] DEBUG  bpws.runtime.bus - Response for external
      invoke is [WSIFResponse:
      serviceID = '{http://tempuri.org/services/CustomerReg}CustomerRegService15d8362-
                1040f701b25--8000'
      operationName = 'authenticate'
      isFault = 'false'
      outgoingMessage = 'org.apache.wsif.base.WSIFDefaultMessage@1c8ee34 name:null
            parts[0]:[JROMBoolean: authRet: true]'
      faultMessage = 'null'
      contextMessage = 'null']
```

The fact that this log event denotes a response from an external service is indicated by the sub string '`Response for external invoke`'. The event receiver identifies this log event as a response from an external service if the following condition holds,

```
Matches($LogString, "Response for external invoke")
```

The regular expressions for extracting the time stamp, event ID, partner ID, operation name and variables are the same as the regular expressions presented for the generic structure of log events above. Following the extraction of the above information, the event receiver is ready to generate monitoring events. As discussed in Section 5.2.2.2 a response to an *invoke* activity is transformed into a conjunction of EC predicates that signify the occurrence of the response to the invocation and initiation of each variable due to the response, the event receiver generates the following two events from this instance of log event,

```
Happens(in:{http://tempuri.org/services/CustomerReg}CustomerRegService15d8362-1040f70
1b25--8000:authenticate(Thread-35), 1116950528236, ℜ(1116950528236, 1116950528236))
Initiates(in:{http://tempuri.org/services/CustomerReg}CustomerRegService15d8362-
 1040f701b25--8000:authenticate(Thread-35), valueOf(authRet, _authRet), 1116950523510)
```

Generation of Monitoring Events for Assign Activities

The log events shown below signify an example of execution of an assign activity.

```
2005-05-24 17:02:08,286  [Thread-34] DEBUG  bpws.runtime.bus - Processing request:
      BPWSBus.BUSRequestAssign assignSrc= com.ibm.cs.bpws.model.FromImpl@1439bd9
      assignDest =com.ibm.cs.bpws.model.ToImpl@aa91ef activity = Activity assign2 with
      parent named assign2 and enclosing scope named assign2 clientData
      com.ibm.cs.bpws.runtime.AssignRT$AssignInfo@fce051 callback = Activity
      carServiceProcess with no parent and no enclosing scope

2005-05-24 17:02:08,306  [Thread-39] DEBUG  bpws.runtime.flow - Processing event:
      [AssignmentValueReadyEvent
      clientData=com.ibm.cs.bpws.runtime.AssignRT$AssignInfo@fce051, value
      [JROMString: loc: One]
```

The first of these log events denotes an *assign* activity due to the presence of the sub string `BPWSBus.BUSRequestAssign` in the *Event_Description*. In this log event, the unique ID `AssignRT$AssignInfo@fce051` identifies the specific instance of the assignment. The name of the *assign* activity appears after the sub string `activity = Activity` and the unique ID appears in the sub string `com.ibm.cs.bpws.runtime.AssignRT$AssignInfo@fce051`.

The *event receiver* identifies this log event as an *assign* activity if the following condition holds

```
Matches($LogString, "BPWSBus.BUSRequestAssign")
```

The sub string that contains the name of the *assign* activity can be extracted by the following expression,

```
SubString($LogString, "activity = Activity [a-zA-Z0-9_f$-]* with")
```

The sub string that contains the unique ID for this *assign* activity can be extracted by the following expression,

```
SubString($LogString, "AssignRT[a-zA-Z0-9$@-]*")
```

The second log event denotes the completion of the *assign* activity. The sub string `AssignmentValueReadyEvent` in the *Event_Description* signifies that this instance of the log event is a completion of an *assign* activity. The unique ID of the *assign* activity is "`AssignRT$AssignInfo@fce051`".

The *event receiver* identifies this log event as the completion of an *assign* activity if the following condition holds,

```
Matches($LogString, "AssignmentValueReadyEvent")
```

The regular expressions for extracting variables are the same as the regular expressions presented for the generic structure of log events above. Following the extraction of the above information, the event receiver is ready to generate monitoring events. As discussed in Section 5.2.2.2 an *assign* activity is transformed into a conjunction of EC predicates that signify the occurrence of the assignment and initiation of each variable due to the assignment, the event receiver generates the following two events from this instance of log event,

```
Happens(as:assign2(AssignRT$AssignInfo@fce051), 1116950528286, ℜ(1116950528286,
1116950528286))
Initiates(as:assign2(AssignRT$AssignInfo@fce051), valueOf(loc, _loc), 1116950528306)
```

Generation of Monitoring Events for *Transition Conditions*

The log event shown below signify an example of execution of a transition condition,

```
2005-05-24 17:02:08,256  [Thread-39] DEBUG  bpws.runtime.flow - about to eval
condition (bpws:getVariableData('authRes' , 'authenticateReturn') = true())
```

This log event denotes a transition condition checking due to the presence of the sub string `"about to eval condition"` in the *Event_Description*. This log event signifies that the variable `authenticateReturn` holds some value and it should be checked now. The *event receiver* identifies this log event as transition condition if the following condition holds,

```
Matches($LogString, "about to eval condition")
```

The regular expressions for extracting the time stamp, event ID are the same as the regular expressions presented for the generic structure of log events above. The variable name is extracted by manipulating substring `bpws:getVariableData('authRes' , 'authenticateReturn')`. With all this information the event receiver generates the following monitoring event

```
HoldsAT(valueOf(authenticateReturn, _ authenticateReturn), 1116950528256)
```

5.2.3.2 IEventReceiver

The interface *IEventReceiver* exposes the following methods:

- `void setEventPatterns(eventPatterns[] Signature)` - The *monitor manager* uses this method to set the event patterns that will determine the runtime events which are relevant to a monitoring activity in the *event receiver*. The parameter *eventPatterns* specifies a list of event patterns where each event pattern is of type *Signature*. The structure of *Signature* is shown below,

Data Structure: Signature		
Purpose: Instance of this type holds the signature of an Event or a Predicate		
Field Name	**Data Type**	**Description**
`_ecName`	`xsd:string`	This field contains the event calculus (EC) name of this event/predicate, e.g. *Happens, Initiates* etc.
`_prefix`	`xsd:string`	This field contains the prefix which signifies the type of this event/predicate, e.g. *in* for invocation, see Section 3.3.3.2 in Chapter 3 for full list of prefix values.
`_operationName`	`xsd:string`	This field contains the name of the operation, in this event/predicate
`_partnerName`	`xsd:string`	This field contains the name of the partner involved in this event/predicate
`_variables[]`	`Variable`	This field contains 0 or more variables associated with the operation
`_fluent`	`Fluent`	This field contains the fluent (if any) associated with the predicate

Data Structure: Variable		
Purpose: Instance of this type holds a variable for an operation		
Field Name	**Data Type**	**Description**
`_name`	`xsd:string`	This field contains the name of the variable
`_type`	`xsd:string`	This field contains the type of the variable, e.g. *string*, *int* etc.
`_value`	`xsd:string`	This field contains the value of the variable

Data Structure: Fluent		
Purpose: Instance of this type holds a fluent of a Predicate		
Field Name	**Data Type**	**Description**
`_firstVariable`	`Variable`	This field contains the first variable of the fluent
`_value`	`Variable/Function`	This field contains the value of the variable. This value can be specified by a variable or a function call.

Data Structure: Function		
Purpose: Instance of this type describes a an external or internal function call		
Field Name	**Data Type**	**Description**
`_operationName`	`xsd:string`	Contains the name of the function
`_parameters[]`	`Variable/ Function`	Contains 0 or more parameters associated with the operation
`_partnerName`	`xsd:string`	Contains the name of the partner that provides the operation, e.g. *self* or some name that signifies external partner

- `void setEventSource(ipAddress xsd:string, port xsd:int)` – The *monitor manager* uses this method to specify the event source to the event receiver. The parameter *ipAddress* indicates the *host* and the parameter *port* indicates the port in that host where the execution environment of an SBS system will write the runtime events that the event receiver will listen to.

- `void startEventReceiver()` – This method is used to start the *event receiver.*

- `void stopEventReceiver()` – This method is used to stop the *event receiver.*

5.2.4 The Simulator

The *simulator* is a component that we have developed in order to generate monitoring events by simulating the execution of BPEL processes. In this section we discuss the mechanism used to implement the simulator with the aid of an example. The simulator offers its functionality through the interface *ISimulator*. We describe the interface *ISimulator* at the end of this section.

The *simulator* takes as input:

(i) the set of all the execution paths of a given BPEL process expressed as event calculus formulas,

(ii) The size n of the possible value set (domain) for each of the non-time variables in the formulas in (i) . The *simulator* generates a domain for each variable, where each domain contains n number of randomly generated values that are used as possible values of the variable during the simulation process.

(iii) A distribution function, called $dist_{exec}$, for the time interval between the execution of two consecutive paths of the BPEL process.

(iv) A distribution function, called $dist_{open}$, for the occurrence time of all constrained predicates in a formula (i.e. execution path) for which one of the lower boundary (LB) or the upper boundary (UB) has not been specified. This distribution function is used to determine the value of the occurrence time of the corresponding predicate. Following formulas are used to determine this value,

Let *offset* is the random number generated by applying $dist_{open}$, then

$$occurrence\ time = LB + offset, \text{ if LB is specified}$$
$$occurrence\ time = UB - offset, \text{ if UB is specified}$$

For the constrained time variables for which both the lower boundary (LB) and the upper boundary (UB) have been specified, the *simulator* applies uniform distribution function, $dist_{uni}$, within $\Re(LB,UB)$ to determine the occurrence time of the corresponding predicate.

Given (i)-(iv), the *simulator*:

- Generates randomly a start time point *ST*
- Selects randomly a formula, F_i, representing an execution path and generates all the events in it in the order they are expected given the time constraints of the formula. More specifically,

- The time stamp of the event that corresponds to the unconstrained predicate in F_i is ST_i. The *simulator* uses the distribution function $dist_{exec}$ to determine the value of ST_i. The following formula is used to generate ST_i,

  Let offset_i is the random number generated by applying $dist_{exec}$ for F_i, then

  $$ST_i = ST_{i-1} + offset_i, \text{ where } ST_0 \text{ is the current system time.}$$

- The time stamp of each of the successive events of F_i is computed based on the lower boundary and upper boundary (i.e. $\mathcal{R}(LB,UB)$) and the distribution function of the time variable of the relevant predicate as described in (iv) above.

- When the *simulator* generates an event for a predicate with a time stamp, it also updates the upper boundary (UB) and lower boundary (LB) of ranges of all the other predicates in the formula that depend on the time variable of this predicate. Also, for the non time variables of the predicate used to generate the event that have not been already assigned some value, the *simulator* selects randomly a value from the domain of the respective variable. The selection of a random value from the domain assumes that different values in the domain have equal probability (i.e. each value has a probability 1/n, where n is the size of the domain) of being selected.

The *simulator* also assigns a unique *id* to each generated event. All the events which are generated in a simulation must have unique *id*s with the exception of the following cases,

(i) Events that instantiate pairs of predicates in a formula that signify a *receive* activity and the corresponding *reply* activity. Such events must have same *id*.

(ii) Events that instantiate pairs of predicates in a formula that signify an invocation of an operation of an external web service and the corresponding response from that web service. These events are assigned the same *id* in order to be able to correlate the invocation with the response.

(iii) Events generated as instances of a predicate that signifies a *receive* activity in a formula and the predicates in the same formula that signify the initiation of fluents to represent the value of the input variable of the operation called by the receive activity. These events must have same *id* in order to be possible to correlate them.

(iv) Events generated as instances of a predicate that signifies response from the execution of an operation in an external web service and all the predicates in the same formula that signify the initiation of fluents to represent the value of the output variable of the relevant operation. These events must have same *id* in order to be possible to correlate them.

After generating all the events in F_i the simulator calculates a random interval *offs* by applying the distribution function $dist_{exec}$. It then randomly selects another formula F_{i+1} and generates all the events in F_{i+1} as described above where the starting time for F_{i+1} is *ST+offs*. The *simulator* repeats these steps as long as the BPEL process is simulated.

```
(F1) ∀ t1:time, _z:string
     Happens(rc:s:O(_ID),t1,R(t1,t1)) ^
     Initiates(rc:s:O(_ID),valueOf(z,_z),t1) ^ (∃ t2)
     Happens(re:s:O(_ID,_z),t2,R(t1,t1 + tu * 100))

(F2) ∀ t1:time, _x:string, _y:int
     Happens(in:p:A(_ID1,_x),t1,R(t1,t1)) ^ (∃ t2)
     Happens(ir:p:A(_ID1),t2,R(t1,t2)) ^
     Initiates(ir:p:A(_ID1),valueOf(y,_y),t2) ^ (∃ t3)
     Happens(in:q:B(_ID2,_x,_y),t3,R(t2,t2 + tu * 50))

** tu = 1 ms.
```

Figure 5.7: Execution paths of a BPEL process expressed as EC formulas

As an example of the event generation mechanism described above consider a BPEL process whose execution paths are the formulas *F1* and *F2* in Figure 5.7. Assuming that, the size of the domain of the variable *_x* is 4, the size of the domain of the variable *_y* is 5, and the size of the domain of the variable *_z* is 5. The *simulator* has generates the following domains:

- domain for *_x* : {aqa, yab, vsbac, zpad}
- domain for *_y* : {348, 9856, 3401, 23, 65923}
- domain for *_z* : {btma, nfbrc, hbwqd, mbsqe, djbfk}

Then, assuming that

(i) The mean and variance of the distribution function $dist_{exec}$ are 0.8 seconds and 0.2 seconds respectively.

(ii) The mean and variance of the distribution function $dist_{open}$ are 0.2 seconds and 0.5 seconds respectively.

the *simulator* generates *ST= 1135694663208* as described above. Subsequently assuming that it randomly selects *F2*, the following events will be generated from *F2*

- *Event 1*

  The first event to be generated from *F2* corresponds to the unconstrained predicate, **Happens**(in:p:A(_ID1,_x),t1,R(t1,t1)) which signifies an invocation of an operation called A in the external web service *p*. The *simulator* randomly picks a value for the variable *_x* from its domain, say *yab*, and assigns a unique *id* to the first event. The time stamp for this event is *ST*. Thus, the first event generated from *F2* is

  Happens(in:p:A(id1,yab), *1135694663208*)

- *Event 2*

  The second event to be generated from *F2* corresponds to the second predicate in *F2*, **Happens**(ir:p:A(_ID1),t2,R(t1,t2)) which signifies the response from the execution of the operation A in the external web service that was invoked by **Happens**(in:p:A(id1,yab), *1135694663208*). The value of the lower boundary *t1* of the second event is set to *1135694663208* (i.e., the same as the time stamp of *event 1*) and the value of the upper boundary *t2* is undefined. The *simulator* uses the distribution function $dist_{open}$ to compute a random number which it subsequently adds to *t1* to determine the value of *t2*. Thus, if the computed random number is 0.004 the *simulator* converts this number into milliseconds ($dist_{open}$ is specified in seconds but the *simulator's* clock operates in milliseconds as t_u=1 ms) and adds it to *t1* to obtain the value of *t2*, i.e. t2 = *1135694663212*. Furthermore, the ID of the event generated for this predicate must be the same as the ID of the *event 1*.Thus, the second event generated from *F2* is

  Happens(ir:p:A(id1), *1135694663212*)

- *Event 3*

  The third event to be generated from *F2* corresponds to the third predicate in the formula, i.e. **Initiates**(ir:p:A(_ID1),valueOf(y,_y),t2). This predicate signifies the initiation of fluent due to the response from an external web service. The second predicate in *F2* signifies the response from the external web service that initiates this fluent. Therefore the ID of the event generated for the third predicate must be the same as the ID

of the *event 2*. The value of t2 is *1135694663212* (due to the *event 2* time stamp). The *simulator* randomly picks a value for the variable *_y* from its domain, let it picks 3401. The third event generated from *F2* is,

Initiates(ir:p:A(id1),valueOf(y,3401), *1135694663212*)

- *Event 4*

The fourth event to be generated from the *F2* corresponds to the fourth predicate in the formula, i.e. **Happens**(in:q:B(_ID2,_x,_y),t3,R(t2,t2 + t_u * 50)). This predicate signifies the invocation of an operation in an external web service. By virtue of the generation of *event 3*, the variable *_x* has been assigned to the value *yab* (due to *event 1*) and the variable *_y* has been assigned to the value *3401* (to *event 3*). The value of the lower boundary of the time variable of the predicate is *1135694663212* as it must be the same as the time stamp of *event 2* and the value of the upper boundary of the time variable of the predicate is *1135694663262* as it must be equal to the timestamp of *event 2* plus 50 time units t_u. After establishing the range of t_3 as $\mathfrak{R}$(*1135694663212, 1135694663262*), the *simulator* creates a random number in this range according to $dist_{uni}$ and assigns it to *t3*. Assuming that the computed random number is *1135694663236*, the fourth event generated from *F2* is,

Happens(in:q:B(id2,yab,3401), *1135694663236*)

When the *simulator* generates the events for all the predicates in *F2*, it randomly selects another formula, say *F1*. To determine the time stamp of the first event generated from *F1* the *simulator* uses the distribution function $dist_{exec}$ to compute a random number and update the value of *ST* by adding the random number to the previous value of *ST*. Assuming that the computed random number is 1.35, the *simulator* converts this number into milliseconds ($dist_{exec}$ is specified in seconds but the *simulator's* clock operates in millisecond as t_u=1 ms) and adds it to the previous value of *ST* to obtain the new value of *ST*, i.e. *ST* = *1135694664558*. Hence the time stamp of the first event generated from *F1* (due to the unconstrained predicate) is *1135694664558*. The subsequent events from *F1* are generated from *F1* following the same mechanisms as in case of *F2* described above. The *simulator* repeats this event generation steps for as long as the simulation of a BPEL process takes place.

5.2.4.1 ISimulator

This interface consists of the following methods:

- `void setExecutionPaths(execPaths xsd:string)` – The *monitor manager* uses this method to set the execution paths in the simulator. The parameter *execPaths* is the string representation of an XML document that contains all the execution paths in a given BPEL process expressed in event calculus. The XML document is written according to the schema presented in Section 3.3.5 in Chapter 3.

- `void setDomains(domains[] Domain)` – The *monitor manager* uses this method to define the variable domains in the simulator. The parameter *domains* signifies a list of domain definition, where each domain definition is of type *Domain*. The structure of Domain is shown below,

Data Structure: Domain		
Purpose: Instance of this type holds a domain for a non time variable		
Field Name	**Data Type**	**Description**
`_varName`	`xsd:string`	This field contains the name of the variable for which the domain is defined.
`_type`	`xsd:string`	This field contains the type of the variable.
`_size`	`xsd:int`	This field defines the number of distinct values of the variable in the domain.

- `void setExecPathDistribution(mean xsd:double, variance xsd:double)` – The monitor manager uses this method to set the distribution function for the execution path interval in the simulator. The parameters *mean* and *variance* hold the mean and variance of the distribution, respectively.

- `void setUnConstrainedTimeDistribution(mean xsd:double, variance xsd:double)` – The monitor manager uses this method to set the distribution function for the unconstrained time variables. The parameters *mean* and *variance* hold the mean and variance of the distribution, respectively.

- `void startSimulator()` – This method is used to start the simulator.

- `void stopSimulator()` – This method is used to stop the simulator.

5.2.5 Event Database Handler

The *event database handler* maintains the communication between the event database and the other components of the monitoring framework. This component receives events from the *event receiver* and stores the events in the event database in the order that they arrive. It also provides access to the events in the event database of the framework. The event database handler is accessible through the *IEventDBHandler* interface.

5.2.5.1 IEventDBHandler

This interface consists of the following methods:

- `void notify(event Event)` – The *event receiver* or the *simulator* uses this method to push an event into the event database. The parameter *event* specifies the event to be stored, which is of type *Event*. The structure of *Event* is shown below,

Data Structure: Event		
Purpose: Instance of this type holds a runtime event		
Field Name	**Data Type**	**Description**
`_signature`	`Signature`	This field contains the event signature
`_timeStamp`	`xsd:long`	This field holds the event occurring time
`_NG`	`Xsd:boolean`	This field takes the value *true* if the event is negated and *false* otherwise.

- `Event getNextEvent()` – The *monitor* uses this method to retrieve the next event from the event database (events are retrieved in the exact order in which they have been recorded).

5.2.6 Formula Database Handler

The *formula database handler* maintains the communication between the formula database and the other components of the monitoring framework. This component receives template instances for a given formula and stores the template instances in the formula database. It also allows to retrieve the template instances for a given formula from the formula database, or update the template instances for a given formula in the formula database. The formula database handler renders its functionality through the *IFormulaDBHandler* interface.

5.2.6.1 IFormulaDBHandler

The interface *IFormulaDBHandler* consists of the following methods:

- `void storeTemplates(templates[] Template, mode xsd:string)` - The *monitor* uses this method to store all the template instances of all the formulas to be monitored in the formula database initially. The parameter *mode* specifies whether these template instances are used for monitoring with respect to recorded events only, or with respect to mixed (both recorded and derived events) events. The parameter *templates* specifies a list of template instances, where each template instance is of type *Template*. The structure of *Template* is shown below,

Data Structure: Template		
Purpose: Instance of this type holds a Template instance		
Field Name	**Data Type**	**Description**
`_id`	`xsd:string`	This field holds the ID of the formula
`_status`	`xsd:string`	This field contains the monitoring decision about this template, e.g. *satisfied, or violated*
`_body[]`	`Predicate/RelationalPredicate`	This field contains the predicates, the time predicates or the relational predicates in the body of the formula
`_head[]`	`Predicate/RelationalPredicate`	This field contains the predicates, the time predicates or the relational predicates in the body of the formula
`_bindings[]`	`Variable`	This field contains the bindings of all the variables in the template
`_dependants[]`	`Dependant`	This field contains a list of dependants, where each dependant holds the ID of a template that depends on this template. See Section 4.3 in Chapter 4 for the definition of formula dependency.

Data Structure: Predicate		
Purpose: Instance of this type holds a Predicate		
Field Name	**Data Type**	**Description**
`_signature`	`Signature`	This field contains the predicate signature
`_truthValue`	`xsd:string`	This field contains the truth value of the predicate, i.e., *true, false* or *UN*
`_quantifier`	`xsd:string`	This field specifies the quantifier of the predicate, i.e., *existential* or *universal*
`_NoQ`	`xsd:Boolean`	This field indicates whether or not the predicate is negated: it takes the vale *true* if the predicate is negated, and the value *false* otherwise.
`_timeStamp`	`TimeVariable`	This field contains a timestamp that indicates when the truth value of the predicate was set.
`_LB`	`TimeExpression`	This field contains the value of the time expression that indicates the earliest time when the predicate may occur. This value is computed from the expression that defines the lower bound of the predicate and the bindings of the variables of this expression.
`_UB`	`TimeExpression`	This field contains the value of the time expression that indicates the latest time when the predicate may occur. This value is computed from the expression that defines the upper bound of the predicate and the bindings of the variables of this expression.
`_source`	`xsd:string`	This field specifies the type of the event that is used to update the truth value of the predicate, i.e., *recorded, derived or negation as failure.*

_pForm	xsd:Boolean	This field specifies if the predicate is a probabilistic predicate (e.g *pHappens, pInitiates* etc). If the predicate is probabilistic it takes the value *true* and if it is not it takes the value *false*.

Data Structure: Dependant		
Purpose: Instance of this type describes a dependant formula		
Field Name	**Data Type**	**Description**
_id	xsd:string	This field contains a formula ID
_signature	Signature	This field contains a signature

Data Structure: RelationalPredicate		
Purpose: Instance of this type holds a relational predicate like $sub(t1,t2) > v_0$		
Field Name	**Data Type**	**Description**
_operand1	Variable/Function	This field contains the first operand in the relation. The operand can be a function call, or a variable.
_operator	xsd:string	This field contains the relational operator involved in the relation, e.g. >, < etc.
_operand2	Variable/Function	this field contains the second operand in the relation. The operand can be a function call, or a variable

Data Structure: TimeVariable		
Purpose: Instance of this type holds a time variable		
Field Name	**Data Type**	**Description**
_name	xsd:string	This field contains the name of the time variable, e.g. *t1*, *t2* etc.
_value	xsd:long	This field holds the value of this time variable

Data Structure: TimeExpression		
Purpose: Instance of this type holds a time relation		
Field Name	**Data Type**	**Description**
_timePoints[]	xsd:string	This field contains the name of the time variables involved in the time expression, e.g. *t1*, *t2* etc.
_operators[]	xsd:string	This field contains the arithmetic operators involved in the time expression, e.g. +, - etc.
_numbers[]	xsd:decimal	This field contains a list of numbers involved in the time expression

- `Templates[] loadTemplates(formulaId xsd:string, mode xsd:string)` – The *monitor* and the *monitor manager* use this method to retrieve all the template instances of a formula from the formula database. The parameter *formulaId* denotes the formula. The parameter *mode* specifies whether these template instances are used for monitoring with respect to recorded events only, or with respect to mixed (both recorded and derived events) events. This method returns a list of template instances.

- `void updateTemplates(templates[] Template, formulaId xsd:string, mode xsd:string)` – The *monitor* uses this method to update all the template instances of a given formula in the formula database. The parameter *templates* specifies a list of template instances, the parameter *formulaId* denotes the formula and the parameter *mode* specifies whether these template instances are used for monitoring with respect to recorded events only, or with respect to mixed (both recorded and derived events) events.

5.2.7 Monitor Manager

The *monitor manager* is the component that has responsibility for the initiation and co-ordination of the monitoring process and the reporting of its results. Once it receives a request for starting a monitoring activity as specified by a monitoring policy, it checks whether it is possible to monitor the requirements specified in this policy given the BPEL process of the service based system that is identified in the policy. If the requested requirements can be monitored, it starts the event receiver to capture events from this environment. It creates template instances for the formulas to be monitored and passes the template instances to the monitor, which has responsibility for detecting whether the formulas are violated. The monitor manager is accessible through the interface *IMonitorManager.*

5.2.7.1 IMonitorManager

This interface consists of the following methods:

- `void setPolicy(policy xsd:string)` - This methods is used by the service based system execution environment or the *Monitoring Console Interface* (on behalf of human user) to set the monitoring policy. The parameter *policy* is the string representation of an XML document that contains the monitoring policy. The XML document is written according to the schema presented in Section 3.2 in Chapter 3.

- `void setFormulas(formulas xsd:string)` - This method is used to set all the formulas to be monitored to the *monitor manager*. The parameter *formulas* is the string representation of an XML document that contains all the formulas to be monitored. The XML document is written according to the schema presented in Section 3.3.5 in Chapter 3.

- `void setMode(mode xsd:string)` - This method is used to set the monitoring mode. The parameter *mode* specifies the monitoring mode i.e. whether the monitoring will be with respect to the recorded events only, or with respect to the mixed (both recorded and derived) events.

- `void setEventSource(ipAddress xsd:string, port xsd:int)` - This method is used to set the event source. The parameter *ipAddress* signifies a host and

the parameter *port* denotes a port in that host where the SBS execution environment writes the runtime event and the event receiver listens to.

- `Templates[] getTemplates(formulaId xsd:string, mode xsd:string)` - The *monitoring console* use this method to retrieve all the template instances of a formula for which monitoring decision has been made. The parameter *formulaId* denotes the formula. The parameter *mode* specifies whether these template instances are used for monitoring with respect to recorded events only, or with respect to mixed (both recorded and derived events) events. This method returns a list of template instances.

- `void setPollingInterval(time xsd:long)` - This method is used to set time interval between the generation of consecutive reports. The parameter *time* specifies the interval in milliseconds.

- `void startMonitoring()` - This method is used to start the monitoring process.

- `void stopMonitoring()` - This method is used to stop the monitoring process.

- `void setSimulatorConfiguration(simConfig xsd:string)` - This method is used to set the simulator configuration parameters in the *monitor manager*. The parameter *simConfig* is the string representation of an XML document that contains the configuration parameters for the simulator. We have defined an XML schema to specify the simulator configuration. Table 5.1 describes the key elements of the schema. The complete schema is shown in Appendix B.

Table 5.1. Textual description of the elements of the simulator configuration schema

Element	Description
<xs:element name="simConfig" type="simConfigType"/>	This is the element that would be used to define the configuration parameters. It has type *simConfigType*.
<xs:complexType name="simConfigType"> <xs:sequence> <xs:element name="processSpecification" type="fns:processSpecificationType"/> <xs:element name="domains" type="domainsType"/> <xs:element name="execPathDistribution" type="distributionType"/> <xs:element	This element defines the structure of the simulator configuration and has the following child elements: (i) an element called *processSpecification* (see Section 3.2 in Chapter 3) of type *processSpecificationType*, which is used to describe the BPEL process to be simulated, e.g. BPEL file name, WSDL file names, (ii) an element of type *domainsType* named as *domains* which is used to define the domains of the non time variables, (iii) an element called *execPathDistribution* of type *distributionType*

name="unconstrainedDistribution" type="distributionType"/> </xs:sequence> </xs:complexType>	which is used to define the distribution function for the execution interval, (iv) an element called *unConstrainedDistribution* of type *distributionType* which is used to specify the distribution function for the unconstrained time variables.
<xs:complexType name="domainsType"> <xs:sequence> <xs:element name="domain" type="domainType" minOccurs="1" maxOccurs="unbounded"/> </xs:sequence> </xs:complexType>	This element specifies a list of domains for the non time variables. It has at least one child of type *domainType* named as *domain* where each child specify a domain for a non time variable.
<xs:complexType name="domainType"> <xs:sequence> <xs:element name="varName" type="xs:string"/> <xs:element name="varType" type="xs:string"/> <xs:element name="size" type="xs:int"/> </xs:sequence> </xs:complexType>	This element defines the domain of a non time variable. It the following child elements: (i) an element called *varName* of type *string* which specifies the name of the variable. (ii) an element called *varType* of type *string* which is used to specify the type of the variable. (iii) an element called *size* of type *int* which is used to specify the size of the domain.
<xs:complexType name="distributionType"> <xs:sequence> <xs:element name="mean" type="xs:float"/> <xs:element name="variance" type="xs:float"/> </xs:sequence> </xs:complexType>	This element defines a normal distribution function through its standard parameters, i.e. its mean and standard deviation.. It has two child elements: (i) an element called *mean* of type *float* which is used to define the mean of the distribution, (ii) an element called *variance* of type *float* which is used to define the variance of the distribution function.

5.2.8 Monitor

The *monitor* processes the events, which are recorded in the *event database* by the event receiver in the order of their occurrence, identifies other expected events that should have occurred but are not recorded, and checks if the recorded and expected events are compliant with the properties which must be monitored for a system. The monitor offers its functionality through the *IMonitor* interface.

5.2.8.1 IMonitor

This interface consists of the following methods,

- `void setMode(mode xsd:string)` - The *monitor manager* uses this method to specify the monitoring mode in the monitor. The parameter *mode* specifies the monitoring mode i.e. whether the monitoring will be with respect to the recorded events only, or with

respect to the mixed (both recorded and derived) events. These two different modes of monitoring are signified by the values "???" and "???", respectively.

- `void setTemplates(templates[] Template)` - The *monitor manager* uses this method to set the empty template instances of the formulas to be monitored. The monitor manager receives the formulas to be monitored through the method *setFormulas* and creates empty templates for each formula and set the templates to the *monitor* using this method. The parameter *templates* specifies a list of template instances, where each template instance is of type *Template*.

- `void startMonitor()` - This method is used to start the monitor.

- `void stopMonitor()` - This method is used to stop the monitor.

5.2.9 Monitoring Console

The *graphical user interface* is the component of the monitoring framework that enables the user to interact with the framework. More specifically, this component allows the end users of the framework to (i) start the monitoring process, (ii) stop the monitoring process and (iii) configure the simulator.

In addition to these functionalities, the monitoring console incorporates (i) an editor that allows the user to define additional functional properties, QoS properties and assumptions about the behaviour of the system to be monitored, (ii) an event viewer that allows the user to view the events that have been stored in the event database, (iii) a monitoring decision viewer that enables the user to browse the monitoring decisions of the formulas and to save monitoring decisions in XML (we have defined an XML schema to represent deviation templates in XML which is given in Appendix C).

5.3 The Prototype of the Monitoring Framework

The components of the monitoring framework that were described in Section 5.2 have been implemented in a prototype. This prototype has been implemented in Java. As stated in Chapter 2, the prototype assumes that the composition process of the service based system to be monitored is specified in BPEL. In our implementation we have used the *bpws4j* process

execution engine to execute the BPEL process. A user manual of this prototype is presented in Appendix G.

In the rest of this section, we describe how the prototype might be used directly by an end user.

The sequence diagram in Figure 5.8 shows a typical user interaction scenario. For simplicity the message parameters are not shown in the sequence diagram.

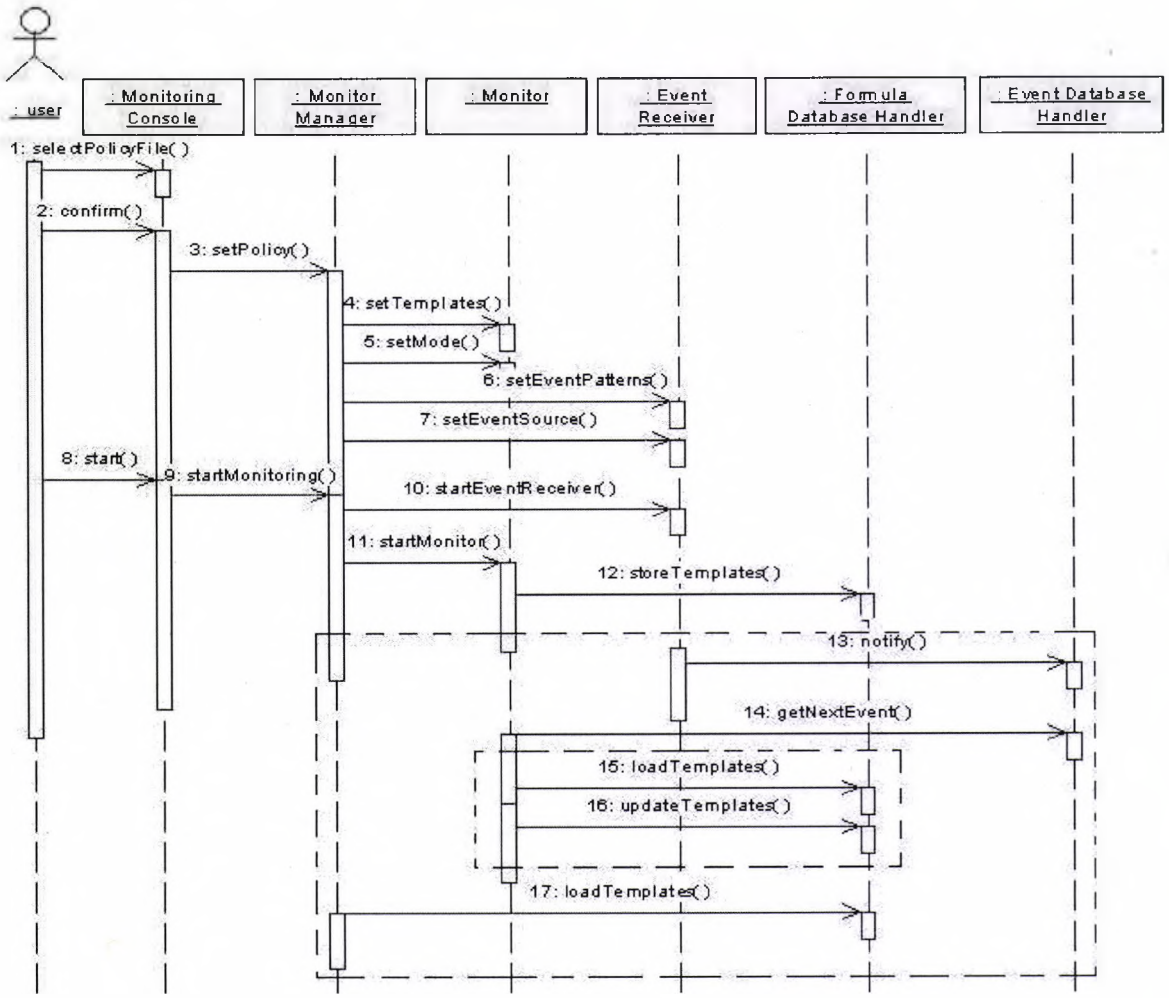


Figure 5.8. A typical user interaction scenario with the monitoring framework

In step 1, the user selects the policy file that contains the formulas and other policy parameters. Figure 5.9 shows a snapshot of the monitoring console. The upper left panel of the console list the ID of the imported formulas and the lower left panel shows the detail of a formula. The monitoring console allows the user to select formulas to be monitored from the

imported formulas. In step 2, the user confirms the selection of formulas. In step 3 monitoring console sets the policy to the monitor manager.

In step 4, the *monitor manager* sets the empty template instances of the formulas to be monitored to the *monitor*. In step 5, the *monitor manager* sets the monitoring mode to the *monitor*. The *monitor manager* sets the acceptable event patterns and the event source to the *event receiver* in the steps 6 and 7 respectively. The user starts the monitoring process in steps 8 and 9 using the *monitoring console*. The *monitor manager* starts the *event receiver* in step 10 and starts the *monitor* in step 11. The *monitor* stores the empty template instances in the formula database in step 12. After starting the monitoring process in step 8, the user should start the BPEL process to be monitored, which is not shown in the sequence diagram.

The steps from 13 to 17, shown in dotted rectangle in the sequence diagram are iterative steps and are repeated as long as the monitoring process continues. In step 13 the *event receiver* writes an event to the event database through the *event database handler*. The *event receiver* reads the runtime events from the *event port* and transforms the runtime event into event calculus form which is not shown in the sequence diagram. The user can view some of the latest events stored in the event database by using the *event viewer*. Figure 5.10 shows a snapshot of the *event viewer*. This step of event viewing is not shown in the sequence diagram for brevity.

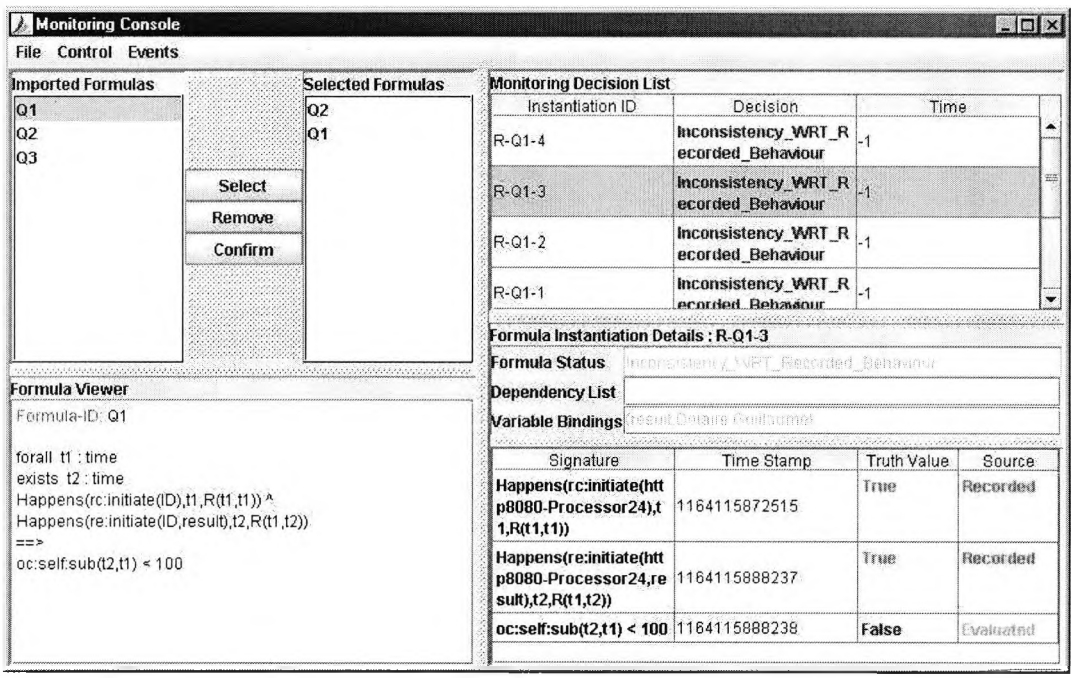


Figure 5.9: Monitoring console

In step 14 the *monitor* picks the next event from the event database through the *event database handler*. Steps 15 and 16 are iterative steps that the *monitor* performs for each event. In step 15 the *monitor* picks all the template instances of a formula from the formula database through the *formula database handler*. The *monitor* process the picked event for each template instance. In step 16 the *monitor* updates the template instances of a formula in the formula database through the *formula database handler*.

In step 17, the *monitor manager* loads the template instances of a formula, creates monitoring decision report for the template instances and sends the monitoring decision report to the *monitoring console*. The *monitoring console* receives the monitoring decision report and updates the display. The upper right panel of the *monitoring console* (see Figure 5.9) list the monitoring decisions. From the monitoring decision list the user can select a particular instance of monitoring decision and the details of the selected monitoring decision is shown in lower right panel of the *monitoring console*.

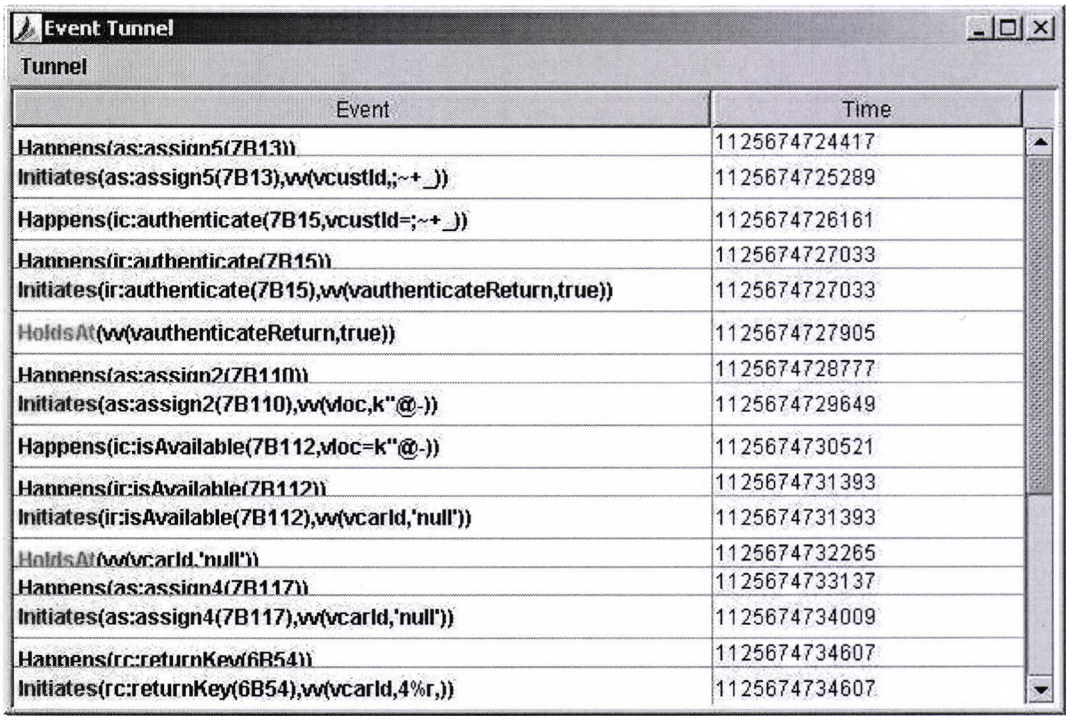


Figure 5.10: Event viewer

Chapter Six

Experimental Evaluation

6.1 Overview

In this chapter, we present a set of experiments that we performed to evaluate the monitoring prototype and demonstrate its applicability to the monitoring of real systems. These experiments were based on two case studies.

In the first case study, we simulated a BPEL process using the simulator that was described in Chapter 5. The objective of this case study was to measure the performance of the monitoring prototype [Mah05]. This performance was measured in terms of: (i) the average delay in processing runtime events, (ii) the time during which the monitor remains inactive whilst engaged in a monitoring session (i.e., the idle time of the monitor), (iii) the average delay that it takes to detect a formula violation following the real occurrence of the violation, and (iv) the number of different types of requirement violations that can be detected at run-time.

The objective of the second case study was dual. Firstly, we wanted to demonstrate the applicability of the monitoring prototype to a simple but real service based system [Spa06]. Secondly, we wanted to evaluate its performance in a real operational context. In this case study, we used a real BPEL process that we developed for the purposes of our experiment and which deployed web-services available on the Internet.

In the following, we discuss the set up of our experiments and the results that we obtained from them.

6.2 Performance Measures

In this section, we define the basic time measures that we used in order to evaluate the performance of the monitoring system.

It should be noted that in general the monitoring system and the system that is being monitored by it (regardless of whether it is based on a really executed or a simulated BPEL process) are two different processes and therefore they have different clocks. Consequently, it is necessary to translate the timelines of these two processes into a common timeline in order to obtain meaningful performance measurements. To achieve this, in our experiments we translated the event times of the system that was being monitored into to monitor's time line. Given this time translation principle, Table 6.1 shows the definitions of the basic time measures that we used in our experiments.

Table 6.1. Basic time measures

Time	Meaning/Calculation
t_i^e	This is the time of occurrence of event *i* as generated by the system being monitored (or the simulator).
t_s^m	This is the starting time of the monitor.
t_c^m	This is the current time of the monitor.
$t_i^{e(d)}$	Time of the recording of an event *i* in the monitor's event database. $t_i^{e(d)}$ is computed by the formula $t_i^{e(d)} = (t_i^e - t_0^e) + t_s^m$ where t_0^e is the time of the occurrence of the first event that was generated by the system that is being monitored (or the simulator).
t_i^m	This is the time when the monitor retrieves an event *i* from its event database to process it.
t_s^{Fj}	Starting time of the decision procedure that the monitor executes to check for violations given the truth values of the predicates in the template *j* of a formula *F*
t_e^{Fj}	This is the completion time of the decision procedure that the monitor executes to check for violations given the truth values of the predicates in the template *j* of a formula *F*

On the basis of the basic time measures shown in Table 6.1, we define the following performance measures:

(i) **Average waiting time of an event:**

The waiting time of an event is the difference between the time point when the event is stored in the event database and the time point when that event is selected by the monitor from the event database for processing. Figure 6.1 illustrates the waiting time of an event.

The average waiting time of events, called *e-delay*, is measured using the following formula,

$$\text{e-delay} = \sum_{i=1,\ldots,K \text{ where } t_i^m - t_i^{e(d)} > 0} (t_i^m - t_i^{e(d)}) \; / \; K$$

where K is the total number of events.

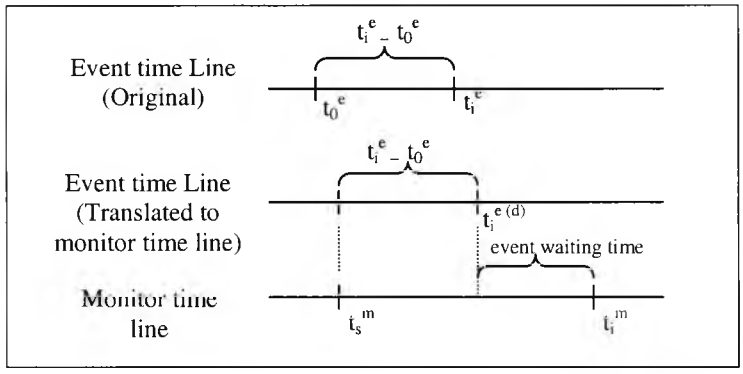


Figure 6.1: Event waiting time

(ii) Monitor's idle time:

The monitor is idle when it has processed all the events that have been stored in the database and gets to a state of waiting for more events to come from the monitored system. Figure 6.2 illustrates the monitor's idle time.

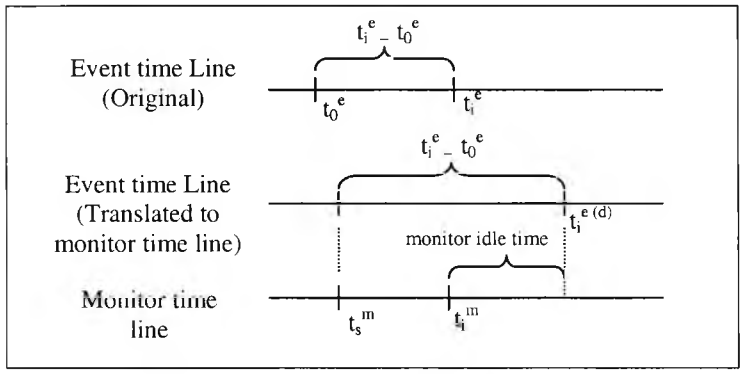


Figure 6.2: Monitor's idle time

The total idle time of the monitor during monitoring session is measured according to the following formula:

$$\text{idle-time} = \sum_{\text{event } i \text{ where } t_i^{e(d)} - t_i^m > 0} (t_i^{e(d)} - t_i^m)$$

(iii) Average decision delay:

The delay in making a decision about a possible violation (or satisfaction) of a formula template is measured as the difference between the time when the monitor makes a decision for the template and the time when the last event e_i that makes it possible to make this decision was recorded in the event database (or the time when the monitor picks up the event e_i from the event database, in case the monitor has idle time). The event that makes it possible

to decide about the satisfaction/violation of a template is the event that is used to update the truth-value of the last predicate in the template. Figures 6.3a and 6.3b illustrate the decision delay.

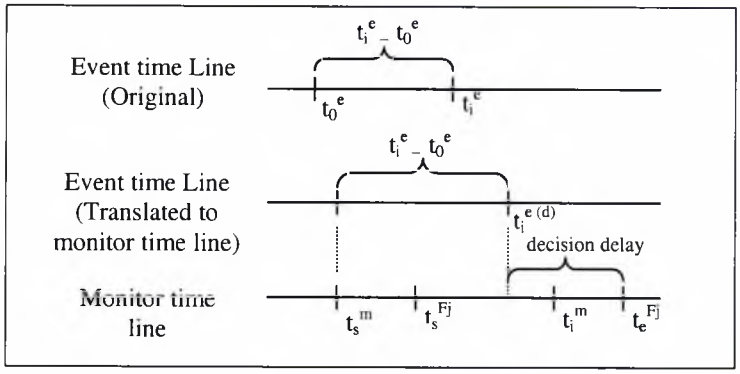


Figure 6.3a: Decision delay

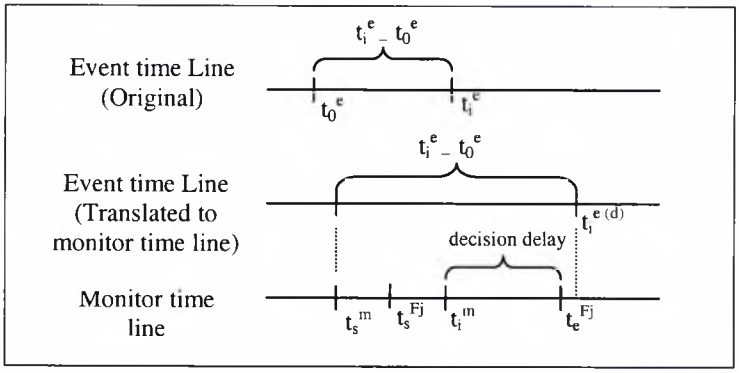


Figure 6.3b: Decision delay

The average delay in making a decision for a template is measured according to the following formula,

$$\text{d-delay} = \sum_{i=1,\ldots,N} d_j / N$$

where

- N is the number of the formula templates for which a decision has been made
- d_j is the delay in making the decision for template j of the formula F that is computed as

  $d_i = t_e^{Fj} - \max_{i \in Fj}(t_i^{e(d)})$ if $t_e^{Fj} - \max_{i \in Fj}(t_i^{e(d)}) > 0$

  $d_i = t_e^{Fj} - \max_{i \in Fj}(t_i^{m})$ otherwise

  where i ranges over the events used to establish the truth values of the formulas in F_j.

6.3 First Case Study: Simulated BPEL Process

In the first case study, we used an implementation of the *Car Rental system* (CRS) that we introduced in Chapter 4. The BPEL specification of the CRS composition process and the WSDL files of the web services deployed by this system are presented in Appendix D. In this case study, we used the *simulator* of the monitoring prototype to generate sequences of events from the BPEL process of CRS. The objectives of this case study were:

(i) To measure the performance of the monitoring prototype in terms of the measures defined in Section 6.2, i.e.
   - Average waiting time for an event
   - Monitor's idle time and
   - Average decision delay.

(ii) To investigate whether performance is affected by the frequency and type of the events which are taken into account, and the size of the domains of the variables used in them.

6.3.1 Experimental Setup

The experiments in this case study were formulated in order to investigate the effect on the performance of the monitoring system of two different factors. These factors were:

- **Average inter arrival time of events**

This factor expresses the average time between the arrival of two events and determines the frequency of event occurrences in a specific time period. In the experiments, we used two categories of the inter event arrival times: (i) a large event inter arrival time (denoted as EL in the following), which was defined to be 1.0 second and (ii) a small event inter arrival time (denoted as ES) which was defined to be 0.15 seconds.

- **The Size of the Domain of Non-time Variables in Formulas**

This factor expresses the number of the distinct elements in the domains of different non time variables in the formulas that were used in the experiments. In the case of CRS , the basic non time variables that were used in the monitored formulas were the variables taking as values customers, cars and car parks. In the experiments we distinguished two different categories of domains: large domains (denoted as DL in the following) and small domains (denoted as DS

in the following). Large domains were defined as domain having at least 4 times more elements than small domains. In our experiments, the DL category was defined to include a set of 200 customers, 80 cars and 12 car parks, and the DS category was defined to include 50 customers, 20 cars and 3 car parks.

In the first case study, we used two different sets of formulas: (i) sets of formulas including *behavioural properties* and *functional properties* (referred to as BF in the following) and (ii) sets of formulas including *quality of service properties*, *functional proerties* and *assumptions* (referred to as QF in the following). The formulas in the set QF includes *quality of service properties* and *assumptions* that invoke internal operations for statistical computation (see *QF1*, *QF2* and *QF3* in Figure 6.4).

Figure 6.4 shows the exact sets of *BF* and *QF* formulas that were used in the first case study.

Behavioural properties and Assumptions (BA)	**Quality of service properties, and Assumptions (QA)**
(BF1) (∀ t1:Time) (∃ t2:Time) **Happens**(rc:enter(ID),t1,R(t1,t1)) ^ **Initiates**(rc:enter(ID),valueOf(loc,vloc),t1) ^ **Initiates**(rc:enter(ID),valueOf(carId,vcarId),t1) ⇒ **Happens**(rc:returnKey(ID),t2,R(t1+1,t1+70000)) ^ **Initiates**(rc:returnKey(ID),valueOf(carId,vcarId),t2) ^ **Initiates**(rc:returnKey(ID),valueOf(loc,vloc),t2) (BF2) (∀ t1:Time) (∃ t2, t3,t4,t5,t6,t7,t8:Time) **Happens**(re:receiveRequest(ID,carId),t1,R(t1,t1)) ^ ¬ **Happens**(rc:depart(ID),t2,R(t1,t1+30000)) ⇒ **Happens**(as:assign11(ID),t3,R(t2,t3)) ^ **Initiates**(as:assign11(ID),valueOf(carId,vcarId),t4) ^ t3 <= t4 ^ **Happens**(as:assign12(ID),t5,R(t4,t5)) ^ **Initiates**(as:assign12(ID),valueOf(loc,vloc),t6) ^ t5 <= t6 ^ **Happens**(ic:makeAvailable(ID,loc,carId),t7,R(t6,t7)) ^ **Happens**(ir:makeAvailable(ID),t8,R(t7,t8)) (BF3) (∀ t1:Time) (∃ t2, t3:Time) **Happens**(rc:receiveRequest(ID),t1,R(t1,t1)) ^ **Initiates**(rc:receiveRequest(ID),valueOf(loc,vloc),t1) ^ **Initiates**(rc:receiveRequest(ID),valueOf(custId, vcustId),t1) ^ **Happens**(re:receiveRequest(ID,carId),t2,R(t1,t2)) ⇒ **Happens**(rc:depart(ID),t3,R(t2+1,t2+30000)) ^ **Initiates**(rc:depart(ID),valueOf(carId,vcarId),t3)	(QF1) (∀ t1:Time) (∃ t2:Time) **Happens**(rc:receiveRequest(ID),t1,R(T1,T2)) ^ **Happens**(re:receiveRequest(ID,carId),t2,R(T1,T2)) ^ **HoldsAt**(valueOf(responseTimes,vresponseTimes),t2) ^ **HoldsAt**(valueOf(responseCount,vresponseCount),t2) ⇒ **Initiates**(rc:receiveRequest(ID),valueOf(responseCount, oc:self:inc(vresponseCount)),t2) ^ **Initiates**(rc:receiveRequest(ID),valueOf(responseTimes [vresponseCount], oc:self:sub(t2,t1)),t2) ^ **HoldsAt**(valueOf(responseTimes,vresponseTimes),t2) (QF2) (∀ t1:Time) **HoldsAt**(valueOf(responseTimes,vresponseTimes),t1) ⇒ oc:self:avg(vresponseTimes[]) < 500 (QF3) (∀ t1:Time) (∃ t2:Time) **Happens**(ic:isAvailable(ID,loc),t1,R(t1,t1)) ^ **Happens**(ir:isAvailable(ID),t2,R(t1,t2)) ⇒ oc:self:sub(t2,t1) < 500 (QF4) (∀ t1:Time) (∃ t2, t3:Time) **Happens**(rc:enter(ID),t1,R(t1,t1)) ^ **Initiates**(rc:enter(ID),valueOf(loc,loc1),t1) ^ **Initiates**(rc:enter(ID),valueOf(carId,vcarId),t1) ^ **Happens**(rc:enter(ID),t2,R(t1+1,t2)) ^ **Initiates**(rc:enter(ID),valueOf(loc,loc2),t2) ^ **Initiates**(rc:enter(ID),valueOf(carId,vcarId),t2)⇒ **Happens**(rc:depart(ID),t3,R(t1+1,t2)) ^ **Initiates**(rc:depart(ID),valueOf(loc,loc1),t3) ^ **Initiates**(rc:depart(ID),valueOf(carId,vcarId),t3)

Figure 6.4: Formulas used in the first case study

In the formula set *BF*, the formula *BF1* is a behavioural property of *CRS*, which signifies if *CRS* is notified of the entrance of a car to a car park, it will wait for 70,000 ms to receive the electronic key of the car. The formula *BF2* is a behavioural property of *CRS* which signifies that if *CRS* replies to a car request with the dispatch of an electronic key of a car, *CRS* will waits for next 30,000 ms for that car to depart the car park. If the car does not depart in the

next 30,000 ms *CRS* marks the car as available in the car registry. The formula *BF3* is a functional property about the expected behaviour of a *CRS* customer. As specified by this formula if *CRS* replies to a car request with the electronic key of a car, the car is expected to leave the car park within 30,000 ms from the release of the car key.

In the formula set *QF*, the formulas *QF1-QF3* specify some quality of service properties of *CRS* and the car information service (*IS*). More specifically, the formulas *QF1* and *QF2* are used to monitor the average response time of *CRS*. The formula *QF1* updates a fluent that is used to keep a record of the response time of CRS for each car request made to it within a given time period and the formula *QF2* specifies that the average of the recorded response times should be less than 500ms. The formula *QF3* is used to monitor the response time of the car information service (*IS*). According to this formula, the response time of *IS* should be less than 500ms. The formula *QF4* is a functional property regarding the correctness of the behaviour of the sensoring services used by *CRS*. According to *QF4*, if a car is sensed to enter a car park at some time *t1* and later at some time *t2* the same car is sensed to enter in the same or a different car park then the departure of the relevant car from the first car park must have also been sensed between the two events that notify the entrance of the car in the car parks.

Table 6.2: Experimental setup for first case study

	EL		ES	
	BF	QF	BF	QF
DL	Exp 1	Exp 2	Exp 5	Exp 6
DS	Exp 3	Exp 4	Exp 7	Exp 8

Table 6.2 summarizes the setup of all the experiments that we performed in the first case study. As shown in the table, we performed 8 different experiments in this case study. In the first experiment (Exp 1) we used large non time variable domains (*DL*), large inter arrival time of events (*EL*), and a set of formulas that contained behavioural properties and functional properties (*BF*). The setup of other experiments can be explained similarly according to the table.

To perform the 8 experiments, we generated four sets of events (one set for each of the four different combinations of the event inter-arrival categories and domain size categories: *EL-DL, EL-DS, ES-DL* and *ES-DS*). Each event set was generated by the simulator using the respective parameter values for the particular combination of categories and contained 30,000 events. Furthermore, in each of the eight experiments we got two sets of results. The first result set was produced by checking for inconsistencies caused by recorded events only. The second result set was produced by checking for inconsistencies caused by both recorded and

derived events. Finally, in each experiment we recorded values of the observed aspects of performance for every 2500 events processed by the monitor.

6.3.2 Results and Analysis

The graphs in Figures 6.5-6.10 and the Tables 6.3a-6.3b summarize the performance measures for each experiment in first case study. Table 6.4 shows the number of the different types of inconsistencies recorded in the experiments of the first case study.

On the basis of the experimental findings shown in these figures and tables, we can summarise the effect of the different factors that we outlined earlier on the performance of the monitoring system as follows:

(i) ***Effect of Event Inter Arrival Time and the Number of Events:***

As we can see from Figures 6.5-6.10 and Tables 6.3a-6.3b, the inter arrival time of events has a significant impact on the performance of the monitor. The average decision delay increases linearly up to a certain number of events and then it starts rising sharply (this is explainable since the worst case complexity of the monitoring algorithm is exponential as explained in Section 4.4.5.3 in Chapter 4). In the experiments with a small inter arrival event time (i.e. a high event occurrence frequency) this point was reached earlier than in the experiments with the larger event inter arrival time. For example in the experiments Exp 1, Exp 2, Exp 3 and Exp 4 the sharp rise in the average decision delay occurred in the range 12,500 - 20,000 events, whereas in experiments Exp 5, Exp 6, Exp 7 and Exp 8 the rise occurred in the range 5,000 - 7,500 events (see Figures 6.5- 6.6 and Tables 6.3a-6.3b).

This observation can also be explained from another viewpoint. Initially, the monitor has some idle time as the number of events which it receives and must process is not relatively large. Thus, the waiting time for each event is minimal. As, however, the number of generated templates increases the processing time of each event increases too since each event needs to be checked for unification with more template instances. Thus, gradually the event processing rate goes down and finally at some point the monitor gets saturated with events. After this saturation point, the monitor does not have any more idle time and the waiting time of events starts rising. The effects of the event inter-arrival time onto the monitor idle time and waiting time of events are shown in Figures 6.7-6.10.

Table 6.3a: Performance measures in each experiment in first case study due to large inter arrival time of events

Events	Exp 1					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.061	2882115.773	0	49.465	2814960.45	0
5000	0.14	10327444.71	0	190.648	9780328.022	0
7500	0.211	20311903.66	0	380.227	18569662.99	0
10000	0.286	30965751.73	0	628.586	26904078.38	0
12500	0.361	40493248.9	0	1059.318	32320706.59	0
15000	0.438	46464113.86	0	1845.821	33121362.05	62.147
17500	105.566	47071501.82	99.773	3205.741	33121362.05	529.567
20000	620.064	47071501.82	647.663	5130.91	33121362.05	1445.76
22500	1598.234	47071501.82	1608.76	7624.294	33121362.05	2775.144
25000	2993.661	47071501.82	2934.739	10607.41	33121362.05	4478.603
27500	4643.431	47071501.82	4585.871	13976.859	33121362.05	6528.235
30000	6493.961	47071501.82	6530.651	17923.682	33121362.05	8959.992
Events	Exp 2					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.091	2903108.925	0	0.052	2967482.305	0
5000	0.203	10607815.82	0	0.129	11045382.17	0
7500	0.31	21368746.06	0	0.226	22502870.91	0
10000	0.421	33578733.95	0	0.333	35551656	0
12500	0.535	45481441.2	0	0.462	48017156.42	0
15000	0.68	55077857.71	0	0.628	57133564.49	0
17500	0.817	59807673.73	0	18.854	59574927.7	15.968
20000	108.557	60025946.19	104.199	400.071	59574927.7	408.983
22500	539.554	60025946.19	525.782	1319.926	59574927.7	1324.945
25000	1204.988	60025946.19	1212.635	2736.749	59574927.7	2772.581
27500	2122.789	60025946.19	2142.972	4701.391	59574927.7	4768.888
30000	3426.149	60025946.19	3306.839	7260.928	59574927.7	7289.779
Events	Exp 3					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.62	2970034.239	0	44.702	2900627.736	0
5000	0.137	10639609.58	0	166.843	10045527.63	0
7500	0.205	20823196.4	0	342.41	18856921.09	0
10000	0.28	31591038.51	0	569.358	26837159.9	0
12500	0.36	41423108.29	0	989.133	31802723.23	0
15000	0.434	47836003.95	0	1744.563	32321143.74	84.393
17500	66.291	48776808.37	63.694	3059.824	32321143.74	583.324
20000	531.397	48776808.37	515.636	4926.657	32321143.74	1502.015
22500	1397.854	48776808.37	1362.438	7266.17	32321143.74	2804.571
25000	2666.998	48776808.37	2588.407	10134.352	32321143.74	4485.137
27500	4343.571	48776808.37	4149.021	13524.908	32321143.74	6526.179
30000	6150.478	48776808.37	6001.84	17452.273	32321143.74	8961.901
Events	Exp 4					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.099	2991158.822	0	0.054	3052624.242	0
5000	0.205	10897512.52	0	0.129	11342948.77	0
7500	0.316	21994543.95	0	0.223	23129116.62	0
10000	0.416	34522692.7	0	0.323	36523840.57	0
12500	0.523	47042735.83	0	0.441	49652203.89	0
15000	0.659	57518657.53	0	0.596	59670215.16	0
17500	0.804	63039965.1	0	9.573	62796987.5	7.829
20000	73.465	63507456.15	66.538	379.108	62796987.5	37.0811
22500	437.443	63507456.15	427.65	1279.196	62796987.5	1272.571
25000	1060.275	63507456.15	1070.094	2763.361	62796987.5	2714.539
27500	1930.594	63507456.15	1965.226	4781.028	62796987.5	4684.202
30000	3063.661	63507456.15	3088.665	7263.333	62796987.5	7173.245

Table 6.3b: Performance measures in each experiment in first case study due to small inter arrival time of events

Events	Exp 5					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.037	307923.089	0	61.044	275903.9	0
5000	15.344	465306.81	14.724	451.172	339821.981	46.259
7500	185.958	465306.81	189.954	1310.369	339821.981	313.111
10000	535.566	465306.81	549.909	2650.789	339821.981	800.484
12500	1184.384	465306.81	1093.423	4420.991	339821.981	1505.768
15000	1896.706	465306.81	1833.395	6592.423	339821.981	2432.167
17500	2781.948	465306.81	2808.069	9435.234	339821.981	3621.689
20000	4121.511	465306.81	4019.521	12992.601	339821.981	5080.348
22500	5476.306	465306.81	5438.54	17063.413	339821.981	6840.573
25000	7003.272	465306.81	7037.616	21539.766	339821.981	8876.867
27500	9055.786	465306.81	8803.648	26647.466	339821.981	11161.308
30000	10671.595	465306.81	10733.564	31843.385	339821.981	13703.162
Events	Exp 6					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.091	254651.462	0	0.054	317950.388	0
5000	69.903	292872.091	68.791	21.781	480198.124	21.238
7500	391.143	292872.091	375.264	258.173	480198.124	256.045
10000	932.147	292872.091	921.24	750.001	480198.124	751.783
12500	1725.364	292872.091	1700.375	1560.068	480198.124	1528.874
15000	2744.56	292872.091	2724.408	2669.75	480198.124	2623.602
17500	4014.723	292872.091	4045.351	4161.121	480198.124	4112.585
20000	5714.931	292872.091	5628.046	6075.385	480198.124	6017.351
22500	7589.652	292872.091	7452.746	8460.236	480198.124	8378.831
25000	9646.611	292872.091	9496.353	11247.675	480198.124	11192.828
27500	11977.893	292872.091	11745.189	14556.032	480198.124	14463.238
30000	14228.259	292872.091	14196.938	18122.003	480198.124	18214.029
Events	Exp 7					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.047	326160.937	0	68.895	295677.663	0
5000	15.215	507422.878	13.174	442.465	382368.841	50.981
7500	210.225	507422.878	203.747	1380.839	382368.841	364.169
10000	585.428	507422.878	604.249	2789.446	382368.841	943.071
12500	1195.021	507422.878	1201.218	4675.733	382368.841	1747.257
15000	2010.527	507422.878	1986.644	6971.834	382368.841	2760.324
17500	3034.482	507422.878	3000.168	9970.169	382368.841	4022.457
20000	4086.692	507422.878	4253.283	13365.995	382368.841	5558.078
22500	5590.255	507422.878	5705.089	17619.066	382368.841	7343.352
25000	7137.676	507422.878	7333.787	22325.482	382368.841	9400.816
27500	8838.639	507422.878	9130.805	27339.526	382368.841	11707.465
30000	11148.466	507422.878	11085.052	33171.973	382368.841	14234.971
Events	Exp 8					
	Recorded			Mixed		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	0.1	268165.522	0	0.054	331075.055	0
5000	57.944	318418.155	62.594	17.389	515709.125	16.904
7500	377.311	318418.155	369.754	262.518	515709.125	253.572
10000	920.477	318418.155	928.822	768.075	515709.125	767.476
12500	1823.895	318418.155	1729.778	1597.268	515709.125	1573.281
15000	2894.617	318418.155	2784.594	2695.178	515709.125	2694.62
17500	4191.911	318418.155	4128.912	4217.42	515709.125	4179.042
20000	5891.241	318418.155	5735.87	6153.664	515709.125	6067.629
22500	7787.654	318418.155	7570.593	8581.609	515709.125	8397.319
25000	9852.252	318418.155	9616.205	11336.809	515709.125	11188.247
27500	12086.883	318418.155	11870.93	14571.92	515709.125	14473.237
30000	14489.932	318418.155	14329.772	18446.884	515709.125	18280.026

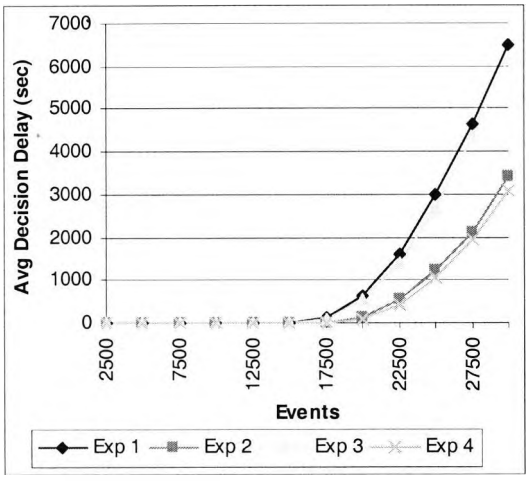


Figure 6.5a: Average d-delay due to recorded events and large inter arrival time of events (EL)

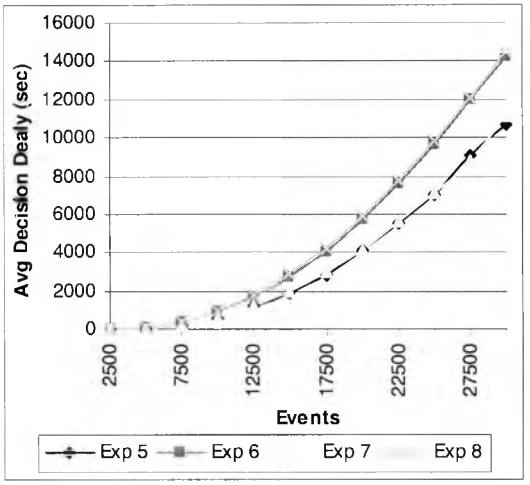


Figure 6.5b: Average d-delay due to recorded events and small inter arrival time of events (ES)

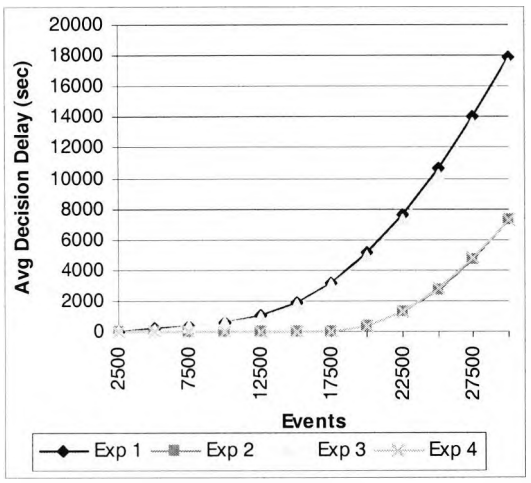


Figure 6.6a: Average d-delay due to mixed events and large inter arrival time of events (EL)

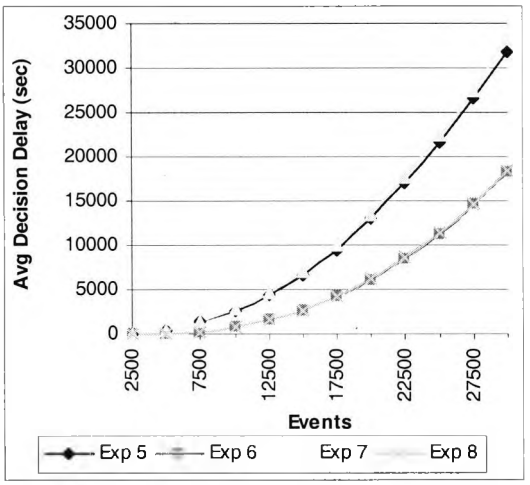


Figure 6.6b: Average d-delay due to mixed events and small inter arrival time of events (ES)

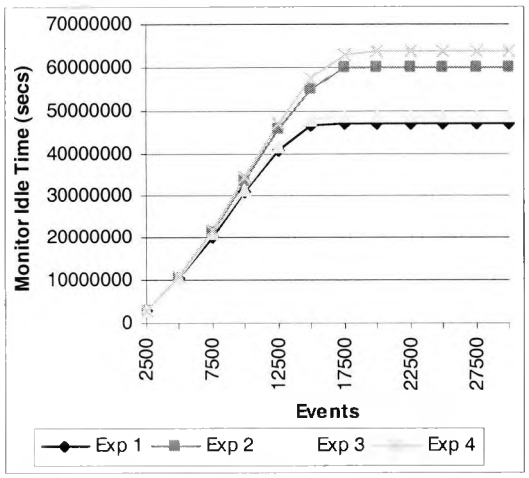


Figure 6.7a: Monitor's idle time due to recorded events and large inter arrival time of events (EL)

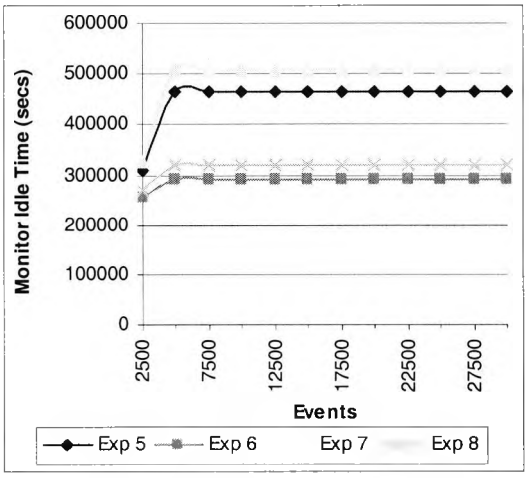


Figure 6.7b: Monitor's idle time due to recorded events and small inter arrival time of events (ES)

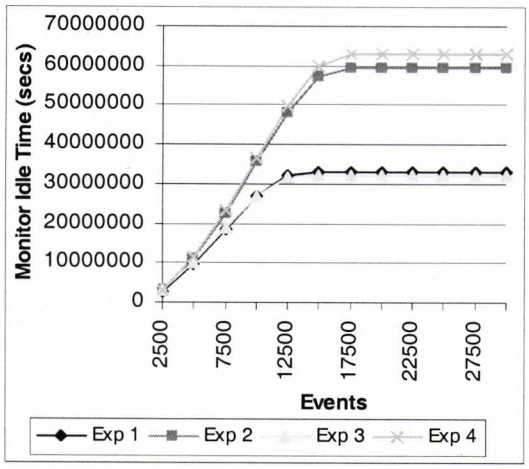


Figure 6.8a: Monitor's idle time due to mixed events and large inter arrival time of events (EL)

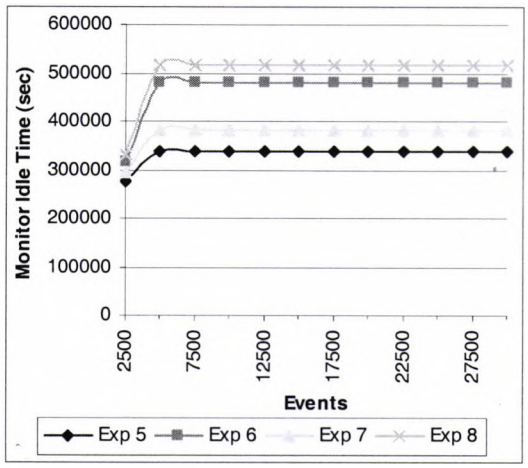


Figure 6.8b: Monitor's idle time due to mixed events and small inter arrival time of events (ES)

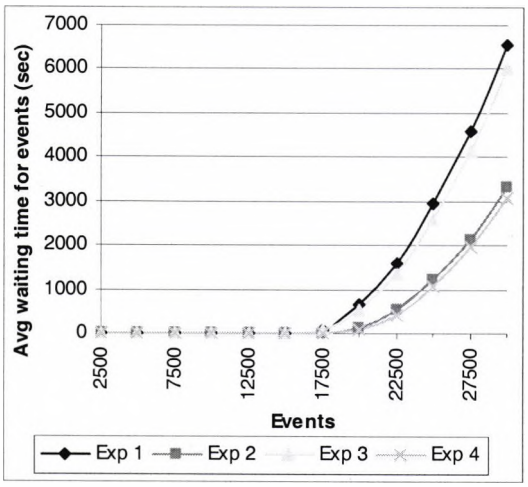


Figure 6.9a: Average waiting time for events due to recorded events and large inter arrival time of events (EL)

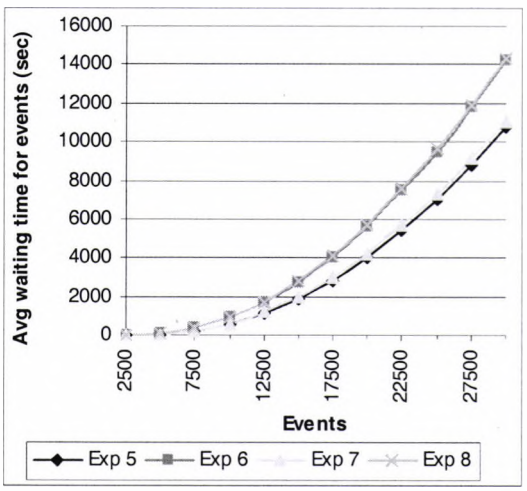


Figure 6.9b: Average waiting time for events due to recorded events and small inter arrival time of events (ES)

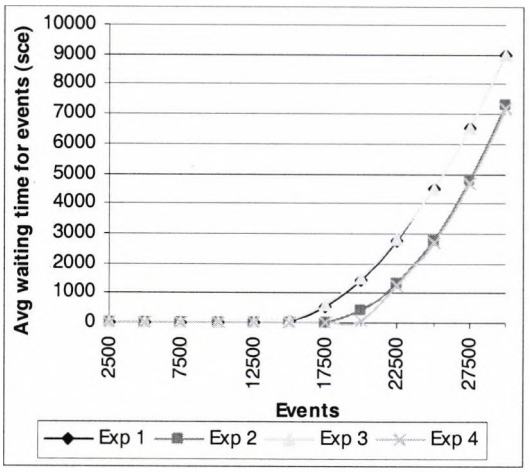


Figure 6.10a: Average waiting time for events due to mixed events and large inter arrival time of events (EL)

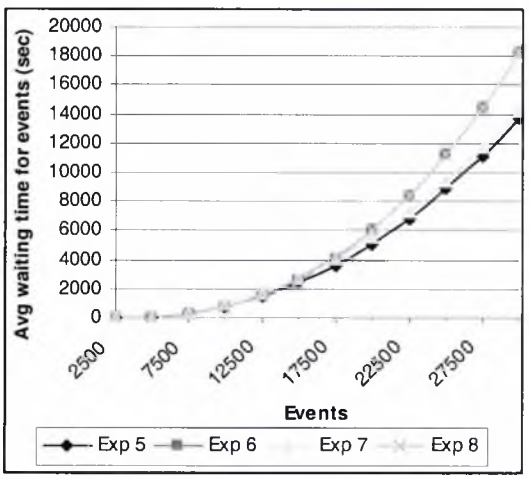


Figure 6.10b: Average waiting time for events due to mixed events and small inter arrival time of events (ES)

In case of small inter arrival time of events the monitor reaches this saturation point earlier than the same in case of large inter arrival time of events. For example, in the experiments with the larger event inter arrival time (i.e. Exp 1, Exp 2, Exp 3 and Exp 4) the monitor reaches the saturation point in the range 15,000 - 17,500 events, whereas in experiments with smaller event inter arrival time (i.e. Exp 5, Exp 6, Exp 7 and Exp 8) the monitor reaches the saturation point in the range 2500 - 5000 events (see Figures 6.7- 6.8 and Tables 6.3a-6.3b). On the other hand, in the experiments with the larger event inter arrival time (i.e. Exp 1, Exp 2, Exp 3 and Exp 4) the waiting time of events start rising in the range 15,000 - 17,500 events, whereas in experiments with smaller event inter arrival time (i.e. Exp 5, Exp 6, Exp 7 and Exp 8) the waiting time of events start rising in the range 2500 - 5000 events (see Figures 6.9-6.10 and Tables 6.3a-6.3b).

Also the inter arrival time of events has major affect on the number of inconsistencies detected. In case of experiments with larger inter arrival time of events (i.e. i.e. Exp 1, Exp 2, Exp 3 and Exp 4) the number of inconsistencies detected is relatively higher than the same in case of experiments with smaller inter arrival time of events (i.e. Exp 5, Exp 6, Exp 7 and Exp 8) (see Table 6.4). This is because, in case of larger inter arrival time of events, the time difference between successive events is large. This fact (the absence of proper event at proper time) causes the occurrence of large number of inconsistencies.

(ii) Effect of Domain Size

As indicated in our experiments the size of the domains of the variables did not have any significant impact on the performance of the monitor. For example, if someone considers the experiment pairs (Exp 1, Exp 3), (Exp 2, Exp 4), (Exp 5, Exp 7) and (Exp 6, Exp 8) whose only difference is the domain size of the non time variables used in the relevant formulas, he/she will realise that the two experiments in each of these pairs had almost the same average decision delay (see Figures 6.5–6.6 and Tables 6.3a–6.3b) and average event waiting time (see Figures 6.9–6.10 and Tables 6.3a–6.3b).

The absence of an effect in this case may not be readily understandable if someone assumes that larger domains of non time variables in formulas would lead to the creation of a larger number of different templates and therefore delays in the monitoring process (i.e., increases in d-delay). It becomes clearer however why this is the case if we consider the following aspects of the monitoring process.

Table 6.4: Number of different types of inconsistencies detected in each experiment in first case study

Events	Exp 1						Exp 2					
	Recorded	Mixed					Recorded	Mixed				
	IRB	IRB	IEB	UB	PIEB	PUB	IRB	IRB	IEB	UB	PIEB	PUB
2500	65	2	0	23	0	33	34	34	98	0	0	0
5000	127	5	0	42	0	66	43	43	201	0	0	0
7500	194	8	0	69	0	112	86	86	302	0	0	0
10000	261	11	0	98	0	146	150	150	404	0	0	0
12500	339	15	0	125	0	178	209	209	509	0	0	0
15000	421	17	0	140	0	203	267	267	619	0	0	0
17500	493	21	0	162	0	234	326	326	725	0	0	0
20000	551	22	0	179	0	262	366	366	823	0	0	0
22500	624	26	0	203	0	308	432	432	924	0	0	0
25000	698	30	0	233	0	344	454	454	1026	0	0	0
27500	767	35	0	259	0	382	486	486	1127	0	0	0
30000	829	39	0	296	0	418	528	528	1221	0	0	0
Events	Exp 3						Exp 4					
	Recorded	Mixed					Recorded	Mixed				
	IRB	IRB	IEB	UB	PIEB	PUB	IRB	IRB	IEB	UB	PIEB	PUB
2500	70	6	0	22	0	26	63	63	100	0	0	0
5000	145	10	0	45	0	57	112	112	202	0	0	0
7500	208	14	0	68	0	86	157	157	301	0	0	0
10000	280	19	0	88	0	105	230	230	397	0	0	0
12500	354	24	0	107	0	132	295	295	498	0	0	0
15000	423	27	0	124	0	153	320	320	600	0	0	0
17500	486	28	0	143	0	182	375	375	701	0	0	0
20000	560	31	0	166	0	210	436	436	802	0	0	0
22500	630	34	0	187	0	226	479	479	899	0	0	0
25000	707	38	0	207	0	255	537	537	1007	0	0	0
27500	788	43	0	231	0	298	595	595	1116	0	0	0
30000	860	47	0	259	0	332	654	654	1217	0	0	0
Events	Exp 5						Exp 6					
	Recorded	Mixed					Recorded	Mixed				
	IRB	IRB	IEB	UB	PIEB	PUB	IRB	IRB	IEB	UB	PIEB	PUB
2500	11	0	0	0	0	0	16	16	98	0	0	0
5000	20	0	0	2	0	11	23	23	202	0	0	0
7500	29	0	0	3	0	20	40	40	300	0	0	0
10000	42	0	0	3	0	42	52	52	397	0	0	0
12500	74	1	0	5	0	54	62	62	505	0	0	0
15000	90	1	0	5	0	66	75	75	608	0	0	0
17500	102	3	0	6	0	75	108	108	713	0	0	0
20000	123	3	0	9	0	97	144	144	811	0	0	0
22500	136	5	0	10	0	101	173	173	912	0	0	0
25000	147	5	0	13	0	118	215	215	1011	0	0	0
27500	173	5	0	14	0	122	245	245	1111	0	0	0
30000	175	5	0	17	0	149	300	300	1209	0	0	0
Events	Exp 7						Exp 8					
	Recorded	Mixed					Recorded	Mixed				
	IRB	IRB	IEB	UB	PIEB	PUB	IRB	IRB	IEB	UB	PIEB	PUB
2500	12	0	0	1	0	4	62	62	99	0	0	0
5000	32	0	0	3	0	19	125	125	206	0	0	0
7500	47	1	0	5	0	32	183	183	312	0	0	0
10000	59	1	0	6	0	36	237	237	417	0	0	0
12500	81	1	0	7	0	41	293	293	516	0	0	0
15000	96	1	0	10	0	59	351	351	611	0	0	0
17500	115	2	0	11	0	63	401	401	722	0	0	0
20000	115	2	0	13	0	89	442	442	826	0	0	0
22500	128	2	0	14	0	96	457	457	934	0	0	0
25000	140	2	0	16	0	104	465	465	1034	0	0	0
27500	142	3	0	19	0	126	476	476	1135	0	0	0
30000	172	3	0	20	0	135	485	485	1244	0	0	0

According to the monitoring algorithm, a new instance of a template is created whenever a new variable in the template is unified with a new value. On the basis of this process, someone may conclude that in a smaller domain we may have small number of templates (because of the repetition of values) and in a larger domain we may have large number of templates (because of more new values of variables), that may affect the performance of the monitor. But this is not always the case, in fact the number of templates does not depend on the size of the domain, it only depends on the number of events being handled by the monitor. Consider the following formula,

Potential Instance Creation point	Formula
(1)	**Happens**(rc:enter(ID),t1,R(t1,t1)) ^
(2)	**Initiates**(rc:enter(ID),valueOf(loc,vloc),t1) ^
(3)	**Initiates**(rc:enter(ID),valueOf(carId,vcarId),t1) ⇒
(4)	**Happens**(rc:returnKey(ID),t2,R(t1+1,t1+70000)) ^
(X)	**Initiates**(rc:returnKey(ID),valueOf(loc,vloc),t2) ^
(X)	**Initiates**(rc:returnKey(ID),valueOf(carId,vcarId),t2)

This formula has four potential points where a new instance of a template for this formula can be created. More specifically,

1. A new instance of the template is created if a new **Happens**`(rc:enter(id),t)` event occurs. At this point instance creation does not depend on the variable value i.e. domain size. Let we have **Happens**`(rc:enter(id1),510)`. Now we have two instances of the template, (i) empty template I1 and (ii) an instance of the template, I2, where the truth value of the first predicate is set to true.

2. A new instance of the template is created from I2, if an **Initiate**`(rc:enter(id),t)` event occurs that corresponds to the previous **Happens**`(rc:enter(id,t))`. Thus assuming that, we have, **Initiates**`(rc:enter(id1),valueOf(loc,l1),510)`, monitor will create a new instance of the template, I3, from the instance I2. At this point more new instances of the template, created from I2, are possible if we have events like,

   Initiates`(rc:enter(id1),valueOf(loc,l2),510)`
   Initiates`(rc:enter(id1),valueOf(loc,l3),510)`
   Initiates`(rc:enter(id1),valueOf(loc,l4),510)`

   In reality, however, it is impossible to have two events having same signature and ID, but different variable values.

Again if we have events like

```
Initiates(rc:enter(id2),valueOf(loc,l2),510)
Initiates(rc:enter(id3),valueOf(loc,l3),510)
Initiates(rc:enter(id4),valueOf(loc,l4),510)
```

There will be no impact on the template instance I2, since the events do not correspond to the previous Happens event due to different IDs. So at this point, the value of the variable (therefore the domain size) doesn't have major impact on the creation of a new template instance.

3. Same as 2.
4. Same as 1.

Therefore the value of the non-time variables (i.e. the domain of the non-time variable values) do not alone drive the creation of new template instances rather it's the *ID* of the event (i.e. total number of different events) together with the value of non-time variables control the creation of new template instances.

In the experiments of first case study, we used two sets of formulas, namely *BF* and *QF*. The formulas in the set *QF* use internal operations for statistical computations. The experimental results does not show any affect of this difference between formulas on the performance (e.g. average decision delay, monitor's idle time and average waiting time for events) of the monitor. But obviously the detection of different types of inconsistencies is dependent on the types of formulas to a great extent. For example, the occurrence of *unjustified behaviour* is only possible to detect if a behavioural property is monitored and that behavioural property is dependent on some other functional property or assumption. Similarly the occurrence of *potentially unjustified behaviour* is only possible to detect if a behavioural property is monitored which is dependent on some other functional property or assumption and the predicate in the behavioural property and the predicate in the functional property or assumption that causes the dependency should have the possibility of having overlapping time ranges at runtime. In the formula set *BF* used in the experiments *BF2* is a behavioural property and it is dependent on *BF3* for the **Happens**`(rc:depart(ID),t2,R(t1,t1+30000))` predicate. Therefore in the experiments based on the formula set *BF* the monitor detected cases of *unjustified behaviour* and *potentially unjustified behaviour* along with *inconsistency of recorded behaviour* . See the inconsistencies for the experiements Exp 1, Exp 3, Exp 5 and Exp 7 in Table 6.4. Again the occurrence of *inconsistency of expected behaviour* is possible to detect if a formula is monitored which is dependent on some other formula. Similarly the occurrence of *possible inconsistency of expected behaviour* is possible to detect if a formula is monitored which is

dependent on some other formula and the predicates that cause the dependency should have the possibility of having overlapping time ranges at runtime. In the formula set *QF* used in the experiments, the *QoS* property *QF2* is dependent on the assumption *QF1* for the predicate `HoldsAt(valueOf(responseTimes,vresponseTimes),t1)` and this predicate does not have any time range. Also the formula set *QF* does not contain any behavioural property. Therefore in the experiments based on the formula set *QF* it is not possible to have occurrences of *unjustified behaviour, potentially unjustified behaviour* or *potential inconsistency of expected behaviour*. See the inconsistencies detected in the experiments Exp 2, Exp 4, Exp 6 and Exp 8 in Table 6.4.

6.4 Second Case Study: Real BPEL Process

In the second case study, we developed a BPEL process, called *Quote Tracker Process (QTP)*. This process allows a user to get a stock quote in US dollars by providing the symbol of a stock traded in the New York Stock Exchange (NYSE) and convert the obtained quote to some other currency. In a typical scenario, *QTP* receives a stock symbol of NYSE from the user, invokes a web service called *Stock Quote Service (SQS)* to get the quote for that symbol and returns the received quote to the user. *QTP* then waits for 30 seconds for the receipt of a country name from the user. If the *QTP* receives a country name within this period, it invokes a second web service, called *Currency Exchange Service (CES)*, to get the currency exchange rate between the US and that country. If *QTP* does not receive a country name from the client within the next 30 seconds, it invokes *CES* to get the currency exchange rate between the US and the UK. Once *QTP* gets the currency exchange rate, it invokes a third web service, called *Simple Calculator Service (SCS)*, to convert the quote into the target currency. In our implementation of the *QTP*, we used the services *SQS* and *CES* provided by XMethods [Xme05] and we implemented *SCS* locally. The BPEL and WSDL files of *QTP, CES, SQS* and *SCS* are presented in Appendix E. The WSDL files of *SQS* and *CES* are also available at http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl and http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl respectively.

The objectives of this case study were to:

(i) Demonstrate the applicability of our monitoring prototype to an operational service based system that deploys third party web services.

(ii) Measure the efficiency of our monitoring prototype in terms of the performance measures defined in Section 6.2, i.e.

- Average waiting time for an event
- Monitor's idle time and
- Average decision delay.

6.4.1 Experimental Setup

It is established from first case study that the *domain size* do not affect the performance of the monitor. So, in this case study, our objective was to investigate the performance of the monitoring system in scenarios with different event inter arrival times. Figure 6.11 shows the formulas used in the experiments of this case study.

```
(F1) (∀ t1: Time)
     Happens(ic:CES:getRate(ID,country2,country1),t1,R(T1,T2)) ^ (∃t2:Time) ^
     Happens(ir:CES:getRate(ID),t2,R(t1,t2)) ^
     Initiates(ir:CES:getRate(ID),valueOf(Result,result1),t2) ^ (∃t3:Time) ^
     Happens(ic:CES:getRate(ID,country2,country1),t3,R(t2,t3)) ^ (∃t4:Time) ^
     Happens(ir:CES:getRate(ID),t4,R(t3,t4)) ^ (t4 ≤ T2)
     Initiates(ir:CES:getRate(ID), valueOf (Result,result2),t4) ⇒ result1 = result2
(Q1) (∀ t1: Time)
     Happens(ic:CES:getRate(ID,country2,country1),t1,R(t1,t1)) ^ (∃t2:Time) ^
     Happens(ir:CES:getRate(ID),t2,R(t1,t2)) ⇒ oc:self:sub(t2,t1) < 1000
(Q2)(∀ t1: Time)
     Happens(ic:SQS:getQuote(ID,symbol),t1,R(T1,T2)) ^ (∃t2:Time) ^
     Happens(ir:SQS:getQuote(ID),t2,R(T1,T2)) ^
     HoldsAt(equalTo(responseTimes,vresponseTimes),t2) ^
     HoldsAt(equalTo(responseCount,vresponseCount),t2) ⇒
     Initiates(ir:SQS:getQuote(ID), valueOf (responseCount,oc:self:inc(vresponseCount)),t2) ^
     Initiates(ir:SQS:getQuote(ID), valueOf (responseTimes[vresponseCount],oc:self:sub(t2,t1)),t2) ^
     HoldsAt(equalTo(responseTimes,vresponseTimes),t2)
(Q3)(∀ t1: Time)
     HoldsAt(equalTo(responseTimes,vresponseTimes),t1) ⇒ oc:self:avg(vresponseTimes[]) < 1000
```

Figure 6.11: Formulas used in the second case study

The formula *F1* in Figure 6.11 was used to monitor the functional consistency of *CES*. More specifically, *F1* specifies, that within a specific time period [T1,T2] *CES* should return the same exchange rate for requests related to the same pair of input countries. The formula *Q1* in the same figure was used to monitor the response time of *CES*, i.e., *Q1* specifies that the response time of any invocation of *CES* should be less than 1000ms. The formulas *Q2* and *Q3* are used to monitor the average response time of the web-service *SQS*. More specifically, *Q2* stores the response time of each invocation of *SQS* within a given time period in the fluent variable *responseTimes*, and *Q3* states that the average of the stored response times should be less than 1000ms.

To deploy *QTP* over a non trivial period of time and with a non trivial level of load, we developed a client program of it which repeatedly calls *QTP* at random intervals. The client program randomly picks up a stock symbol from a set of 15 symbols and calls *QTP* for the quote of this symbol. When it receives a quote from *QTP*, the client program randomly picks up a country name from a set of 15 country names and calls *QTP* with this country name, to convert the quote in the latter country's currency. Subsequently, the client program sleeps for a random time interval and then repeats the above sequence of actions with different sets of data. The random sleeping interval of the client program was used in the experiments to regulate the inter-arrival time of the events exchanged between the deployed web-services and *QTP*.

Based on the sleeping interval of the *QTP* client program and the type of events used for monitoring we performed the following four experiments.

1. Experiment 1 (*Exp-1*) – In this experiment we: (i) set the sleeping interval of the *QTP* client to a random number that is uniformly distributed between 0 and 5, i.e. the *QTP* client program sleeps a random interval of up to 5 seconds between the successive calls that it makes to *QTP*, and (ii) used only the recorded events to monitor the formulas shown in Figure 6.11.

2. Experiment 2 (*Exp-2*) – In this experiment, we: (i) set the sleeping interval of the *QTP* client programme to a random number that is uniformly distributed between 0 and 5, and (ii) used both the recorded and the derived events to monitor the formulas shown in Figure 6.11.

3. Experiment 3 (*Exp-3*) – In this experiment, we: (i) set the sleeping interval of the *QTP* client programme to a random number that is uniformly distributed between 0 and 10,and (ii) used only the recorded events to monitor the formulas shown in Figure 6.11.

4. Experiment 4 (*Exp-4*) – In this experiment, we: (i) set the sleeping interval of the *QTP* client programme to a random number that is uniformly distributed between 0 and 10, and (ii) used both the recorded and the derived events to monitor the formulas shown in Figure 6.11.

Table 6.5 summarises the experimental setup of the experiments in this case study.

Table 6.5: Experimental setup for the second case study

	Exp 1	Exp 2	Exp 3	Exp 4
Sleeping interval of QTP client	Uniformly distributed in the range [0,5] secs	Uniformly distributed in the range [0,5] secs	Uniformly distributed in the range [0,10] secs	Uniformly distributed in the range [0,10] secs
Event type	Recorded	Recorded+Derived	Recorded	Recorded+Derived
Average inter-arrival event time (s)	0.215	0.211	0.39	0.383
Total event time span in secs/hours	4301.131/~1.194h	4230.421/~1.175h	7816.67/~2.171h	7666.825/~2.129h

6.4.2 Results and Analysis

The graphs in Figures 6.12-6.14 and the Table 6.6 summarise the performance measures for each experiment in second case study. Table 6.7 shows the number of different types of inconsistencies recorded in the experiments of the second case study.

Although we tried to vary the inter arrival time of the events using the sleeping interval of the *QTP* client programme, the inter arrival times of the events in experiments Exp 1, Exp 2, Exp 3 and Exp 4 did not differ that much due to the randomness of the sleeping interval. The inter arrival times of the events in Exp 3 and Exp 4 are slightly higher than those of in Exp 1 and Exp 2, but much lower than the large inter arrival time used in first case study.

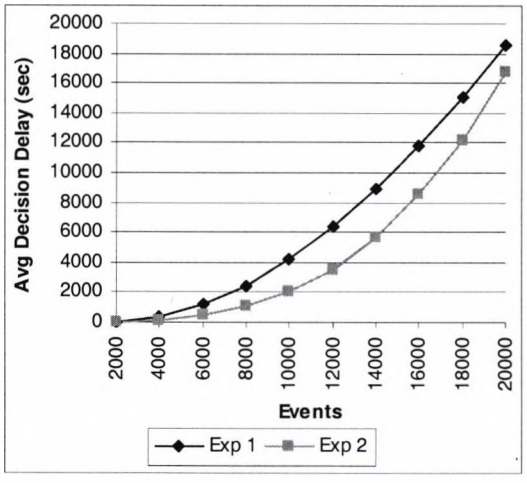


Figure 6.12a: Average d-delay due to small inter arrival time of events

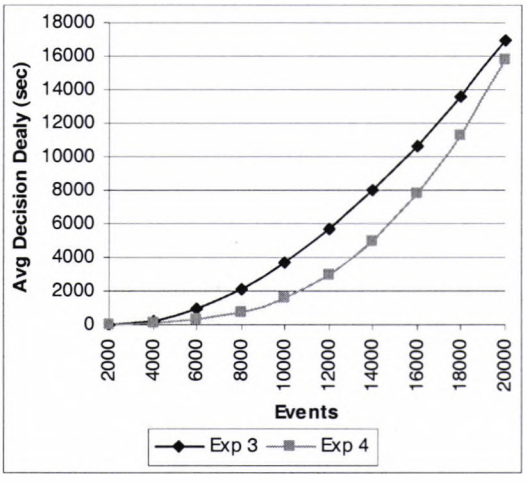


Figure 6.12b: Average d-delay due to large inter arrival time of events

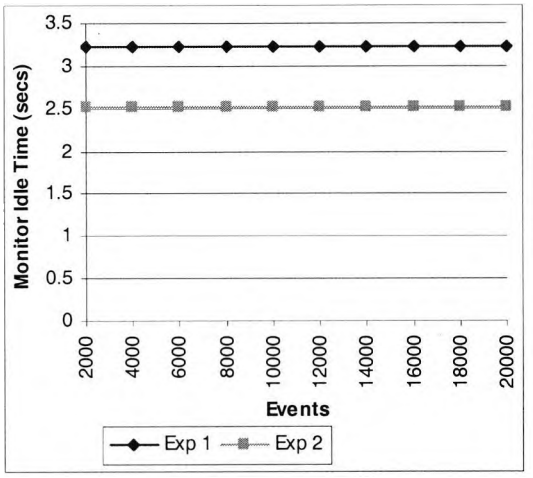


Figure 6.13a: Monitor's idle time due to small inter arrival time of events

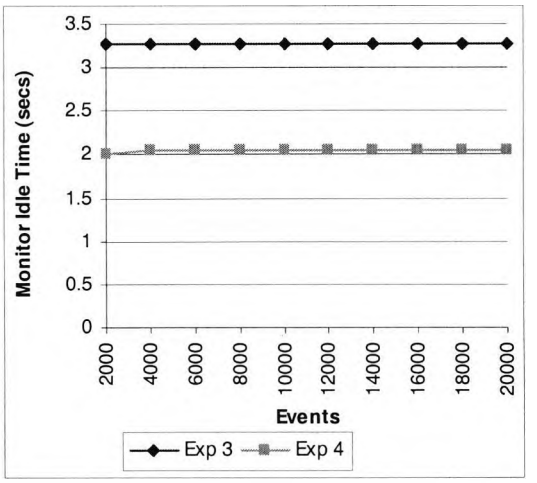


Figure 6.13b: Monitor's idle time due to large inter arrival time of events

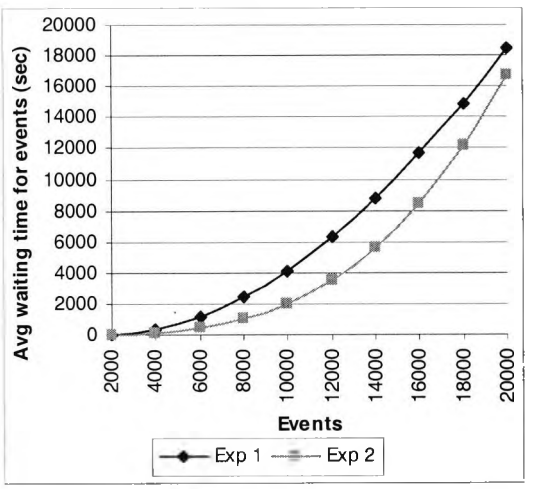


Figure 6.14a: Average waiting time for events due to small inter arrival time of events

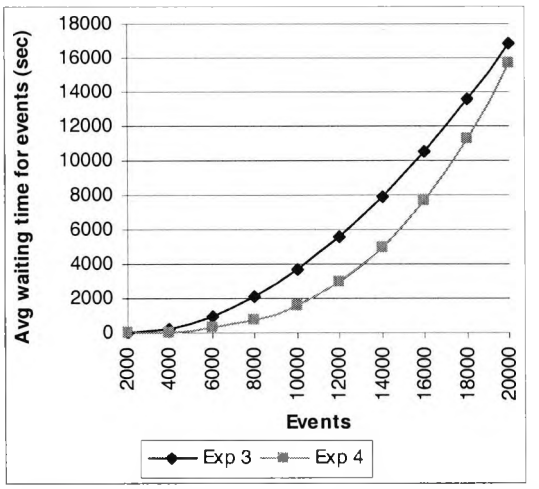


Figure 6.14b: Average waiting time for events due to large inter arrival time of events

From the experimental setup shown in Table 6.5, the inter arrival times in all the four experiments of the second case study (i.e. Exp 1, Exp 2, Exp 3 and Exp 4) were very close to the small inter arrival time used in first case study (i.e. Exp 5, Exp 6, Exp 7 and Exp 8). Comparing the performance measures of the second case study with those of the first case study, it is evident that in the experiments (i.e. Exp 1, Exp 2, Exp 3 and Exp 4) in second case study, the average decision delay and average waiting time of events were longer than the average decision delay and event waiting time in the corresponding experiments (i.e. Exp 5, Exp 6, Exp 7 and Exp 8) in the first case study, respectively. Also the monitor reached the saturation point lot earlier in the experiments of the second case study than it did in the corresponding experiments in the first case study. This is because in the second case study we monitored a real BPEL system where the monitoring event extraction by the event receiver and the monitoring were carried out in parallel on the same machine unlike the first case

study in which we monitored a simulated process where the events had been generated by the simulator before the monitoring process started.

Table 6.6: Performance measures in each experiment in second case study

Events	Exp 1			Exp 2		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	33.786	3.226	31.453	18.296	2.515	17.34
5000	358.86	3.226	348.036	154.553	2.515	151.346
7500	1171.223	3.226	1147.333	486.903	2.515	480.631
10000	2457.539	3.226	2419.389	1062.297	2.515	1052.559
12500	4189.215	3.226	4137.165	2007.408	2.515	1987.485
15000	6351.502	3.226	6285.829	3484.586	2.515	3465.397
17500	8894.096	3.226	8813.91	5626.282	2.515	5598.503
20000	11781.145	3.226	11686.918	8510.141	2.515	8461.414
22500	15004.451	3.226	14897.188	12202.262	2.515	12133.4
25000	18568.231	3.226	18448.063	16742.784	2.515	16675.461
27500	33.786	3.226	31.453	18.296	2.515	17.34
30000	358.86	3.226	348.036	154.553	2.515	151.346
Events	Exp 3			Exp 4		
	Avg-D (s)	Idle-T(s)	Wait-T(s)	Avg-D (s)	Idle-T(s)	Wait-T(s)
2500	7.763	3.27	5.858	2.805	2.008	2.157
5000	217.991	3.27	210.827	52.852	2.06	50.456
7500	921.919	3.27	903.032	277.041	2.06	271.23
10000	2118.978	3.27	2080.109	752.986	2.06	742.372
12500	3721.454	3.27	3672.74	1599.769	2.06	1582.662
15000	5671.009	3.27	5611.495	2971.856	2.06	2939.581
17500	7977.397	3.27	7907.862	4985.46	2.06	4945.301
20000	10634.126	3.27	10554.574	7737.525	2.06	7687.654
22500	13620.368	3.27	13531.493	11305.892	2.06	11244.707
25000	16944.98	3.27	16845.583	15752.381	2.06	15678.492
27500	7.763	3.27	5.858	2.805	2.008	2.157
30000	217.991	3.27	210.827	52.852	2.06	50.456

Table 6.7 shows the number of different types of inconsistencies detected in the second case study. The number of the detected inconsistencies in the second case study is in line with the observation of the first case study, that is larger inter arrival time of events causes larger number of inconsistencies than small inter arrival time of events. For example, in Exp 3 and Exp 4 slightly more number of inconsistencies are detected than in Exp 1 and Exp2 as the inter arrival times of the events in Exp 3 and Exp 4 are slightly larger than the inter arrival time of the events in Exp 1 and Exp 2.

It should be noted that in the second case study , we did not monitor any behavioural property of *QTP*. Therefore it was not possible to have cases of *unjustified behaviour* and *potentially unjustified behaviour*. Given the formulas in Figure 6.11, only the formulas *Q2* and *Q3* were dependent on each other (*Q3* is dependent on *Q2* due to the predicate **HoldsAt**`(valueOf(responseTimes,vresponseTimes),t1)` and as this predicate did not have any time range, it was not possible to have *potential inconsistencies of expected behaviour*. Therefore in the experiments of the second case study, we only had the

occurrence of *inconsistency of recorded behaviour* and *inconsistency of expected behaviour* as it is shown in Table 6.7.

Table 6.7: Number of different types of inconsistencies detected in each experiment in second case study

Events	Recorded		Mixed									
	Exp 1	Exp 3	Exp 2					Exp 4				
	IRB	IRB	IRB	IEB	UB	PIEB	PUB	IRB	IEB	UB	PIEB	PUB
2000	14	31	26	7	0	0	0	39	35	0	0	0
4000	48	59	58	7	0	0	0	67	35	0	0	0
6000	83	99	87	7	0	0	0	106	143	0	0	0
8000	115	142	124	7	0	0	0	147	143	0	0	0
10000	154	184	152	7	0	0	0	184	143	0	0	0
12000	196	247	187	7	0	0	0	226	143	0	0	0
14000	229	301	219	7	0	0	0	267	143	0	0	0
16000	279	342	256	7	0	0	0	298	143	0	0	0
18000	321	392	298	7	0	0	0	323	143	0	0	0
20000	350	447	338	7	0	0	0	361	143	0	0	0

6.5 Applicability of the Framework

Our framework supports monitoring of wide range of software properties. Although in Chapter 2, we classified monitorable properties as behavioural properties, functional properties and QoS properties, these types of properties fall under the category *System Capabilities* discussed in [Whi95], or *functional* and *non-functional requirements* discussed in [Abr04]. In [Abr04] *non-functional requirements* are further classified as *performance requirements, maintainability requirements, safety requirements* and *reliability requirements.* We believe EC (hence EC-Assertion) is expressive enough to express these types of properties that enables the monitoring of these types of properties in our framework. However, the experimental results demonstrate that the average delay in the detection of a property deviation might not be negligible. Decision delays may limit the applicability of our framework to monitoring only certain types of properties where the timeliness in the detection of a deviation is not critical for a system (e.g., monitoring of long term performance properties of a system) and exclude time critical properties (e.g. safety). Also, as discussed in Chapter 4, it should be noted, that the current implementation of the framework has some limitations and it can not monitor the properties expressed as past formulas. Regardless, however, of the properties being monitored it should be appreciated that since monitoring takes place in parallel with the operation of a service based system without affecting its performance, it can detect useful deviations from requirements at no significant cost for the system.

Regarding the usability of the framework one may raise the concern that it may not be very easy for a business analyst or a developer to express monitorable properties in EC-Assertion due to the formality of the event calculus first order temporal logic language that underpins it. This concern can be addressed by (i) defining a language that hides the formalism as much as possible and is easy to understand, or (ii) developing patterns for specifying generic monitorable properties in this language and offering them to developers who wish to specify their monitorable properties along with an editor to support the automatic generation of instances of these patterns [Dwy98, Spa07]. Although, EC-Assertion does not raise the level of abstraction at which such properties can be specified and remains close to event calculus adding only special types of events and fluents, it should be noted that it enables the expression of a wide range of monitorable properties as mentioned above. Given our experience in the specification of monitorable properies in EC-Assertion, we believe that following some basic training in basic event calculus, it is possible to write monitorbale properties in EC-Assertion for a given BPEL process specification in relatively short time.

As discussed in Chapter 5, because of modular design of the architecture of the monitoring framework, the monitor can be deployed as a web service (see Appendix G for details). In such scenario, the monitor accepts as inputs, (i) the properties to be monitored expressed in EC-Assertion and (ii) monitoring events expressed in XML according to the schema presented in Appendix G. These two inputs are completely independent of the language used to express the service based system specification and the engine used to execute the service based system. Although in this thesis we adopted BPEL as the service based system specification language and bpws4j as the service based system execution engine, this discussion reveals that our monitor can be used for any other service based system specification language, more specifically any other web service flow specification language (e.g. OWL-S, WSCI, WS-CDL) and any other service based system execution engine e.g. ActiveBPEL, Oracle BPEL Process Manager. To use our monitor with a service based system specification language other than BPEL, a mapping between the language and EC-Assertion should be defined to express the monitorable properties for the specification in EC-Assertion. Similarly to use our monitor with a service based system execution engine other than bpws4j, an event receiver should be developed to capture monitoring events from the service based system execution engine and send the monitoring events to the monitor.

Chapter Seven

Conclusions and Future Works

7.1 Overview

In this final chapter of the thesis, we provide an overview of the research work that resulted in the monitoring framework which was presented in the earlier chapters. We also point out the main novelties of this framework and the contributions that our research has made to the state of the art. Our claims are founded on a comparison with other approaches to service based systems monitoring. We also point out the limitations of our research and discuss directions for future work upon our framework and the more general area of monitoring service based systems.

7.2 Summary of the Work

In this thesis, we have presented a framework that we have developed to monitor the runtime behaviour of service-based systems against a set of requirements specified for these systems. This work has followed a survey of the state to the art in software monitoring which identified limitations of existing monitoring techniques in monitoring requirements for service based systems. The developed framework support requirements specify:

(i) behavioural properties of service based systems,

(ii) functional properties for the individual services of service based systems or groups of such services (e.g. pre-conditions and post-conditions that must be satisfied before and after the execution of operations of individual services), or

(iii) quality of service (QoS) properties, which express the quality requirements of service based system, individual services or group of services.

In addition to behavioural properties, functional properties and quality of service properties, the framework supports the specification of assumptions which are used to generate additional information about the expected service behaviour and its effect on the state of the system.

As part of the development of this framework, we have defined a language for specifying the behavioural, functional and QoS properties of the systems to be monitored as well as and assumptions regarding these systems, called EC-Assertion. This language has its formal foundation in event calculus [Sha99].

The behavioural properties are initially extracted from the specification of the composition process of a service-based system that is expressed in BPEL. This ensures that these properties are expressed in terms of events occurring during the interaction between the composition process and the constituent services of the system to be monitored that can be detected from the log of the execution. The functional, and QoS properties to be monitored and the assumptions which can be used during monitoring are subsequently defined in terms of the identified detectable events by system providers.

Through monitoring scenarios that we have presented in this thesis, we have argued about the necessity of introducing types of runtime deviations from requirements which go beyond classical inconsistencies. And to support these types, we have developed appropriate reasoning mechanisms. These mechanisms have been built upon techniques developed for integrity checking in temporal deductive databases [Ple93, Cho95]. The main difference from these techniques is the formula checking scheme that we deploy and the absence of distinction between integrity constraints and deductive rules in our framework. To be precise in our framework assumptions are used as deductive rules only, but functional properties are treated in both capacities.

This thesis has also proposed an architecture for the implementation of the monitoring framework and discussed mechanisms/algorithms for implementing different components of the architecture. Finally an implementation of the monitoring framework has been presented and its effectiveness has been evaluated through case studies.

7.3 Contributions

The main contributions of this research are summarised in the following:

- **Development of a non intrusive monitoring framework for service based systems implemented in BPEL**

We designed a monitoring framework that can be used to monitor the requirements of service-based systems which are implemented by composition processes expressed in BPEL.

The runtime monitoring of service based system has drawn attention of the research community recently, but only few of the existing strands of work in this area that we have found in the literature focus on monitoring of BPEL processes (e.g. [Bar04a, Bar05a, Bar05b]). Furthermore, most of the approaches in this area focus mainly on monitoring one type of properties, e.g. QoS properties of web services [And04a, And04b, Lud04], functional properties [Rob03a, Rob03b], or the composition of web services and the management of such compositions [Nak01, Nak02, Pic02, Ali03, Zho03, Tos01, Laz04, Laz06a, Laz06b]. The monitoring framework that is described in this thesis is generic enough to support monitoring of a wide spectrum of monitorable properties (see Chapter 6) including functional properties and non functional properties (e.g. safety properties, performance properties, security properties, reliability properties). Furthermore, most of the complexities which arise in case of service based system (see Chapter 2) are not addressed by the different approaches that we identified in the literature. Our monitoring framework has been designed with the objective to support *non intrusive monitoring*, which excludes approaches which perform monitoring by weaving code that implements the required checks inside the code of the system that is being monitored (e.g. monitoring oriented programming [Che03, Che04, Che05b] or approaches that perform service based systems monitoring by weaving code into BPEL processes [Bar04a]). Non intrusive monitoring also excludes approaches which, despite deploying external entities in order to perform the required checks, require the instrumentation of the source code of the monitored system in order to generate the runtime information that is necessary for the checks (e.g. [Fea98, Rob02, Kan00, Kim01, Din02]).

Although in this thesis we chose BPEL as the service based system specification language, it is discussed in Chapter 6 that our approach can be applied to other service based system specification languages like OWL-S, with minimal effort.

- **Detection of new types of property deviations**

The monitoring framework that we have developed supports the detection of five different types of property deviations that may occur during the operation of service-based systems. These deviations are:

(i) Inconsistencies with respect to recorded behaviour
(ii) Inconsistencies with respect to expected behaviour
(iii) Unjustified system behaviour
(iv) Possible violations of behavioural properties and functional properties
(v) Potentially unjustified system behaviour

The above types of deviations are based on the distinction that is made by our monitoring framework regarding the type of events which can be used in order to detect deviations. These events may be of two types: (1) recorded events which have been captured during the operation of the system at runtime or (2) derived events which are generated from recorded events by deduction. If monitoring is based only on recorded events, it can detect only deviation type (i). On the other hand, if monitoring is based on both recorded and derived events, our framework can detect deviation types (ii)-(v) in addition to deviation type (i).

To the best of our knowledge while other approaches support the detection of property deviation of the types (i) above [Fea95, Fea98, Coh97, Din02, Rob02, Rob03a, Rob03b], they do not support the detection of inconsistencies of types (ii)-(v). The detection of the latter types of deviations is however important during the operation of service based systems, as we have argued in Chapter 4, of this thesis, and to this end the framework that is presented in this thesis fills an important gap in this area.

- **Property specification language**

We have defined a language, which has its formal foundation on event calculus, to specify requirements of service based systems expressed as BPEL process. The selection of event calculus as the formal base of our language over other temporal logic languages (e.g. LTL, CTL, and PTL) has been discussed in Chapter 3. The choice of event calculus as the language for specifying the requirements to be monitored against the behaviour of service based systems has been motivated by the need for: (a) expressing the properties to be monitored in a formal language allowing the specification of temporal constraints and (b) being able to monitor an agreement using a well defined reasoning process based on the inference rules of first-order logic (this criterion has also led to the choice of event calculus instead of another temporal logic language). Our language extends standard event calculus by defining special events and fluents which enable the specification of monitorable properties using full first-order logic formalism as well as conditions about time. Another extension with respect to standard event calculus is the use of internal and external operations in formulas which enable the delegation of complex computations of complex data functions to computational entities which are external to the main reasoning engine. The use of such computations (e.g. computation of the standard deviation of a series of values) is often required for the specification of QoS properties. And the delegation of such computations to entities outside the main reasoning engine which checks whether the property is satisfied is important for reducing the cost of the actual computation and making easier the specification of the relevant

properties. Finally, it should be noted that we have defined the property specification language as an XML schema that makes our language applicable to other standards such as WS-Agreement as means of specifying requirements.

- **Implementation of a prototype supporting the monitoring of service based systems implemented in BPEL**

To realise our framework, we have developed a prototype implemented in Java. This prototype provides supports for automatic monitoring of service based systems. More specifically, (i) it automatically extracts the behavioural properties to be monitored from the service composition specification, (ii) it allows users to define additional functional properties, assumptions and QoS properties about the service based system and the services that it deploys, and (iii) it monitors the target service based system automatically driven by the behavioural properties, functional properties, assumptions and QoS properties. The prototype can be used as a stand alone monitoring tool. However, it can also be deployed as a web service. In either case the tool is easy to set up and provides maximum user flexibility.

7.4 Limitations of the Proposed Approach

The framework that we have developed for runtime monitoring of service based systems, has also a number of limitations. These limitations are overviewed below:

- In Chapter 4 we presented the formal analysis of the monitoring scheme. This analysis revealed that the worst case complexity of our monitoring scheme is exponential with factor, $F_n * (2^{E_n}-1) * O(n^2m)$, where F_n is the number of formulas, E_n is the number of events and n is the number of predicates in a formula and m is the number of variables in a predicate. Although our monitoring framework is able to monitor a wide range of properties as listed in Section 7.3, this detection delay may limit the applicability of our framework as discussed in Chapter 6. More specifically our approach would be feasible to monitor only certain types of properties where the timeliness in the detection of a deviation is not critical for a system (e.g., monitoring of long term performance properties of a system) and exclude time critical properties (e.g. safety, reliability).

- The monitoring scheme that was presented in Chapter 4 does not completely realise the conceptual design of our monitoring framework. For example, the monitoring scheme

does not apply abductive reasoning to generate derived events and cannot monitor past formulas.

- It should be noted that, although the property specification language in our framework is expressive enough to support a wide spectrum of monitorable properties, we appreciate that the use of the language for the specification of such properties may be difficult for users who are not familiar with formal languages.

7.5 Plans for Future Work

Future work on the framework that has been introduced in this thesis will focus on the following aspects of the framework:

- Enhancement of the Monitoring Performance of the Framework.
- Support for Property Specification.
- Integration of the Proposed Monitoring Framework to Existing Standards.

7.5.1 Enhancement of Monitoring Performance of the Framework

From the complexity analysis in Chapter 4 and the evaluation results that we presented in Chapter 6, it emerged that the average delay in the detection of a requirement deviation grows exponentially along with the number of events which occur within a system. Although this complexity allows the application of our monitoring framework to cases where the timeliness in the detection of a deviation is not critical (e.g. reliability), it restricts its applicability in cases where this is not the case (e.g. safety). This observed complexity makes it difficult for someone to use our framework in order to monitor time-critical properties like safety and raises the issue of enhancing the performance of our monitoring framework. Given the evaluation results and the current state of the monitoring framework, the enhancement of the monitoring performance of our framework is possible through the following two ways.

(i) One factor that affects the monitoring performance is the number of formulas to be monitored. The *monitor manager* may deploy more than one monitor where each monitor monitors a very small number of formulas. Whilst the distribution of formulas to different monitors is easy in cases where monitoring is concerned with the detection of formula violations by recorded events only, the distribution of formulas to different

monitors needs some static reasoning before the monitoring process starts This is important in order to identify groups of formulas with no dependencies between them and distribute them to different monitors. For example in our framework we apply deductive and abductive reasoning to generate derived events, and these derived events are used to update the templates that are dependent on these derived events. So interdependent formulas should be deployed on the same monitor

(ii) Comparing the results of the second case study with those of the first case study, it is evident that in the second case study the average decision delay is longer than the average decision delay in the first case study. This is because in the second case study, we monitored a real BPEL system where the extraction of monitoring events and the monitoring itself were carried out in parallel on the same machine. But in the first case study, we monitored a simulated BPEL process where the events were generated by the simulator before the monitoring process started. This finding suggests that the monitoring performance can be improved if the monitoring event extraction and the monitoring are organised as separate processes and are possibly deployed on separate computers.

7.5.2 Support for Property Specification

As we discussed earlier, due to the physical distribution of the individual services of a service based system and/or network communication delays, it is impossible to predict the precise time of the response of the component services in service based systems. Even the prediction about the response time of component services is not straightforward. This raises an issue that should be the subject of future investigation. This issue is the development of support for specifying realistic time constraints of the time variables in the formulas based on analysis of time delays in execution histories. This analysis would help the system provider to specify (i) a suitable value for the minimum time t_u between the occurrence of two events, and (ii) more realistic QoS properties which express the quality requirements of individual services or group of services.

Moreover, as we have pointed out in Section 7.4 above, as a formal language, the event calculus based language of our framework is inherently difficult to use by users who do not have training in temporal logic languages. A possibility to address this limitation, would be to develop patterns for specifying generic monitorable properties in this language and offer them to system providers who wish to specify their monitorable properties and an editor to support

the automatic generation of instances of these patterns for specific service based systems. Several efforts have been found in the literature that attempt to develop patterns to specify generic properties for temporal logic languages (see [Dwy98, Spa07] for example). The existence of such patterns and the similarity of the temporal languages which are the focus of the patterns developed so far with event calculus makes it reasonable to believe that such patterns will be possible to develop for our language and motivates this direction of possible future work. A critical aspect for the effectiveness of the pattern-based approach to property specification will be the composability of any developed patterns as this could enable the easier specification of complex properties by composing atomic properties.

7.5.3 Integration of the Monitoring Framework with Existing Standards

As our monitoring framework stands now it can be easily incorporated into standard monitoring frameworks like *WS-Agreement*. As we discussed in Chapter 2, *WS-Agreement* does not support the specification of policies determining the deployment context in which the provision of services of a service based system will be monitored, who will have responsibility for providing the information that is necessary for assessing whether the guarantee terms of the agreement are satisfied and where the results of monitoring will be reported in cases where agreement monitors should actively report deviations from the terms of an agreement rather than waiting to be asked if a deviation has occurred. Furthermore, *WS-Agreement* does not specify a language for defining the service description and guarantee terms of an agreement and an operation protocol that would enable the monitoring of an agreement.

Our framework provides an exact way to address these limitations of *WS-Agreement* [Mah07]. More specifically, as we discussed in Chapter 3, our framework supports the specification of a monitoring policy specifying the composition process of the service based system that is to be monitored, the services deployed by this system, the source of the runtime information which will enable the monitoring of the agreement, the way in which this monitoring is to be performed including the mode, regularity and timing of monitoring. The specification of these aspects of a monitoring agreement is quite important and can be integrated with the context specification of WS-Agreement. Also our EC based property specification language can be used to specify the service guarantee terms of a WS-Agreement (currently, *WS-Agreement* does not offer a language for the specification of these terms).

References

[Abi97] Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J. "The Lorel Query Language for Semistructured Data". International Journal on Digital Libraries. 1 (1997) 68--88

[Abr04] Alain Abran (ed.), "Guide to the Software Engineering Body of Knowledge", IEEE Computer Society, 2004.

[Act05] ActiveBPEL, The Open Source BPEL Engine, July 2005, http://www.activebpel.org/info/news.php

[Aie05] Marco Aiello, Ganna Frankova and Daniela Malfatti, "What's in an Agreement? A Formal Analysis and an Extension of WS-Agreement", Technical Report # DIT-05-039, April 05.

[Ale02] Alessandra Russo, Rob Miller, Bashar Nuseibeh, and Jeff Kramer, "An Abductive Approach for Analysing Event-Based Requirements Specifications". Proceedings of 18th International Conference on Logic Programming, Copenhagen, Denmark, 29 July-1 August 2002, Springer

[Ali03] Ali ShaikhAli, Omer Rana, Rashid Al-Ali and David W. Walker. "UDDIe: An Extended Registry for Web Services". In Proceedings of the Service Oriented Computing: Models, Architectures and Applications, SAINT-2003 IEEE Computer Society Press. Oralndo Florida, USA, January 2003.

[Amo05] Marcelo d'Amorim and Klaus Havelund, "Event-Based Runtime Verification of Java Programs", In 3rd International Workshop on Dynamic Analysis (WODA'05). St. Louis, USA, May 2005.f

[And04] A. Andrieux, C. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, M. Xu. WebServices Agreement Specification (WS-Agreement). Version 1.1, Draft 20, June 6th 2004.

[And04a] Andrew D. H. Farrell, Marek Sergot, Mathias Salle and Claudio Bartolini., David Trastour, Athena Christodoulou, "Performance Monitoring of Service-Level Agreements for Utility Computing Using the Event Calculus", HPL-2004-20R1

[And04b] Andrew D. H. Farrell, Marek Sergot, Mathias Salle and Claudio Bartolini. "Using the event calculus for performance monitoring of Service Level Agreements in Utility Computing". In Proc. Workshop on Contract Languages and Architectures (CoALa2004), 8th International IEEE Enterprise Distributed Object Computing Conference, Monterey, September 2004.

[And05] Andrew D. H. Farrell, Marek Sergot, Mathias Salle and Claudio Bartolini.

"Using the event calculus for tracking the normative state of contracts. " International Journal of Cooperative Information System 4(2--3). June-September 2005.

[Ari96] ARIANE 5, Flight 501 Failure, Report by the Inquiry Board, Paris, 19 July 1996

[Asp03] Aspectj project. http://eclipse.org/aspectj/. 2003.

[Axi03] Axis, http://ws.apache.org/axis/index.html. June 16, 2003

[Bal02] Thomas Ball, Sriram K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis", POPL 2002, January 2002.

[Ban03] A. K. Bandara, E. C. Lupu, and A. Russo. "Using Event Calculus to Formalise Policy Specification and Analysis". In Proceedings of 4th IEEE Workshop on Policies for Distributed Systems and Networks Policy 2003.

[Bar01] M. M. Detlef Bartetzko, Clemens Fischer and H. Wehrheim. "Jass - java with assertions". In Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers, 2001.

[Bar04a] Luciano Baresi, Carlo Ghezzi and Sam Guinea. "Smart Monitoring for Composed Services". International Conference on Service Oriented Computing, November 15-19, 2004, New York, USA.

[Bar04b] Ian Barland, "Promela and SPIN Reference", version 1.3, August 12, 2004

[Bar05a] Luciano Baresi, Sam Guinea and Pierluigi Plebani, "WS-Policy for Service Monitoring", **6**th VLDB Workshop on Technologies for E-Services (TES-05), September 2–3, 2005, Trondheim, Norway

[Bar05b] Luciano Baresi, and Sam Guine, "Towards Dynamic Monitoring of WS-BPEL Processes". Proceedings of the 3rd International Conference of Service-oriented Computing (ICSOC'05). Amsterdam, The Nederlands, 2005. Lecture Notes in Computer Science, volume 3826, pages 269 - 282.

[Bar05c] L. Baresi and S. Guinea."Dynamo: Dynamic Monitoring of WS-BPEL Processes". Proceedings of the 3rd International Conference of Service-oriented Computing (ICSOC'05). Amsterdam, The Nederlands, 2005. Lecture Notes in Computer Science, volume 3826, pages 478 - 483.

[Bel00] P. Bellini, R. Mattolini, and P. Nesi, "Temporal Logics for Real-Time System Specification", ACM Computing Surveys, Vol. 32, No. 1, March 2000

[Ber01] T. Berners-Lee, J. Hendler, and O. Lassila. "The Semantic Web". Scientic Ameri-can, 284(5):34{43, 2001.

[Ber03] A. Bertolino, "Software Testing Research and Practice", Invited presentation at 10th International Workshop on Abstract State Machines ASM 2003, Taormina, Italy, March 3-7, 2003, LNCS 2589, p. 1-21

[Bob01] Jerry Bobrow, "Cliffs Quick Review Algebra I", ISBN: 0-7645-6370-X, May 2001

[Bpe03] Business Process Execution Language for Web Services, version 1.1, May 05, 2003.

http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[Bpw03] Business Process Execution Language for Web Services Java Runtime, BPWS4J, April 30, 2003, http://alphaworks.ibm.com/tech/bpws4j

[Bro02a] Mark Brorkens and Michael Moller. "Jassda Trace Assertions, Runtime Checking the Dynamic of Java Programs". In: Ina Schieferdecker, Hartmut Konig and Adam Wolisz (Eds.), Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems, Berlin, March 2002, pp. 39–48.

[Bro02b] Mark Brorkens and Michael Moller. "Dynamic Event Generation for Runtime Checking using the JDI". In Klaus Havelund and Grigore Rosu (Eds.), Proceedings of the Federated Logic Conference Satellite Workshops, Runtime Verification. Electronic Notes in Theoretical Computer Science 70.4, Copenhagen, July 2002.

[Bow95] Jonathan Bowen, Michael j Hinchey, "Seven More Myths of Formal Methods", IEEE Software, July 1995.

[Cap01] L. Capra, W. Emmerich, and C. Mascolo. "Reflective middleware solutions for context-aware applications" – Lecture Notes in Computer Science, 2192, 2001

[Caz98] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. "Architectural reflection: Bridging the gap between a running system and its architectural specification". In Proceedings of 6th Reengineering Forum (REF'98), pages 12-16, Firenze, Italia, March 1998. IEEEF.

[Che03] F. Chen and G. Rosu. "Towards monitoring-oriented programming: A paradigm combining specification and implementation." In Runtime Verification, 2003.

[Che04] F. Chen, M. Amorim, , and G. Ro¸su. A Formal Monitoring-based Framework for Software Development and Analysis. In Proceedings of the Sixth International Conference on Formal Engineering Methods (ICFEM'04), 2004.

[Che05a] Feng Chen and Grigore Rosu, "Java-MOP: A Monitoring Oriented Programming Environment for Java", TACAS'05, LNCS 3440, pp 546-550. 2005.

[Che05b] Feng Chen, Marcelo d'Amorim and Grigore Rosu, "Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP", RV'05, ENTCS 144, issue 4, pp 3-20. 2005.

[Cho95] Chominski J. "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding", ACM Transactions on Database Systems, 1995

[Cla78] K. L. Clark. "Negation as failure". In H. Gallaire and J. Minker, editors, Logic and Databases, pages 293--322. Plenum Press, New York, 1978

[Cla86] Clarke, E. M., Emerson, E. A., and Sistla A. P."Automatic verification of finite-state concurrent systems using temporal logic specifications". ACM Trans. Program. Lang. Syst. 8, 2 (Apr. 1986), 244–263. 1986.

[Cla96] Edmund M. Clarke and Jeannette M. Wing. " Formal Methods: State of the Art and

Future Directions". ACM Computing Surveys, 1996

[Cla94] Edmund M. Clarke, Orna Grumberg and Long D, "Model Checking", Proceedings: International Summer School on Deductive Program Design, 1994. Springer-Verlag Nato Asi, Series F, Vol. 152, 1996.

[Cla00] Lori A. Clarke, Leon J. Osterweil. "Continuous Self-Evaluation for the Self-Improvement of Software". Springer Verlag Lecture Notes in Computer Science #1936, Proceedings of the 1st International Workshop on Self-Adaptive Software (IWSAS 2000), pp 27–29, April 2000, Oxford, England.

[Cla05] Daniela Barreiro Claro, Patrick Albers and Jin-Kao Hao, " Approaches of Web Services Composition, Comparison between BPEL4WS and OWL-S", In Proceedings of International Conference on Enterprise Information Systems, May 23 – 27, Miami–USA 2005

[Cli03] CLiX, Constraint Language in XML. http://www.clixml.org/spec.html, 2003

[Coh97] Don Cohen, Martin S Feather, K. Narayanswamy & Stephen S. Fickas. "Automatic Monitoring of Software Requirements". Proceedings of the 19th International Conference on Software Engineering, Pages 602 – 603, IEEE Press, 1997

[Coh98] Geoff Cohen, Jeff Chase and David Kaminsky. "Automatic Program Transformation with JOIE". In proceedings of the 1998 USENIX Annual Technical Symposium.

[Col98] Michael Collins, "Formal Methods", Departmental Report, Electrical and Computer Engineering Department, Carnegie Mellon University, Spring 1998

[Con91] Console, L., Dupre, D. T., and Torasso, P., "On the Relationship between Abduction and Deduction". Journal of Logic and Computation 1(5):661—690, 1991

[Cop03] David Coppit. "Engineering Modeling and Analysis: Sound Methods and Effective Tools". PhD thesis, The University of Virginia, Charlottesville, Virginia, January 2003.

[Cor04] Common Object Request Broker Architecture (CORBA), http://www.omg.org/technology/documents/corba_spec_catalog.htm

[Cra93] D. Craigen, S. Gerhart, T. Ralston. An International Survey of Industrial Applications of Formal Methods:Volume 1 – Purpose, Approach, Analysis and Conclusions. Technical Report NISTGCR 93/626, National

Institute of Standards and Technology, Gaithersburg, USA, March 1993.

[Dam02] The DAML Service Coalition. "DAML-S Semantic Markup for Web Services". International Semantic Web Conference (ISWC), 2002.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal Directed Requirements Acquisition", Science of Computer Programming, Vol. 20, 1993, 3-50.

[Dar96] Robert Darimont and Axel van Lamsweerde. "Formal Refinement Patterns for Goal-

Driven Requirements Elaboration". Proceedings of 4th ACM Symposium on the Foundations of Software Engineering (FSE4), San Francisco, Oct 1996, 179–190.

[Dav02] David Daly, Gautam Kar and William H, Senders. "Modeling of Service-Level Agreements for Composed Services". Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications, October 21-23, 2002.

[Del04] Nelly Delgado, Ann Quiroz Gates and Steve Roach. "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools", IEEE Transactions On Software Engineering, VOL. 30, No 12, December 2004.

[Den95] Marc Denecke, "A terminological interpretation of (Abductive) Logic Programming", Lecture Notes In Computer Science; Vol.928, Pages: 15-28, 1995

[Den96] Marc Denecker, Kristof Van Belleghem, Guy Duchatelet, Frank Piessens, Danny De Schreye, "Realistic Experiment in Knowledge Representation in Open Event Calculus: Protocol Specification", In Proceedings of the Joint International Conference and Symposium on Logic Programming, pages 170-184, 1996.

[Die99] F. Dietrich, J.-P. Hubaux, "Formal Methods for Communication Services", Institute for Computer Communications and Applications, Swiss Federal Institute of Technology, CH-1025 Lausanne, August 1999

[Din02] Andrew Dingwall-Smith, Anthony Finkelstein. "From Requirements to Monitor by Way of Aspects". 1st International Conference on Aspect-Oriented Software Development, April 22-26, 2002.

[Dwy98] Dwyer, M.B., Avrunin, G.S. and Corbett, J.C.: Property Specification Patterns for Finite state Verification. Proc. Of 2nd Work. on Formal Methods in Software Practice, (1998)

[Efs02] Christos Efstratiou, Adrian Friday, Nigel Davies, and Keith Cheverst. "Utilising the event calculus for policy driven adaptation on mobile systems". In Jorge Lobo Bret J. Michael and Naranker Duray, editors, *3rd International Workshop on Policies for Distributed Systems and Networks*, pages 13-24, Monterey, Ca., U.S., 2002. IEEE Computer Society.

[Eit95] T. Eiter and G. Gottlob. "The complexity of logic-based abduction", *Journal of the ACM*, *42*(1), 3-42. 1995.

[Eli06] Elisabeth A. Strunk, M. Anthony Aiello, John C. Knight, Eds. "A Survey of Tools for Model Checking and Model-Based Development", Technical Report CS-2006-17 Department of Computer Science University of Virginia June 2006

[Far05] Roozbeh Farahbod, Uwe Glässer, Mona Vajihollahi, "A Formal Semantics for the Business Process Execution Language for Web Services", WSMDEIS 2005: 122-133

[Fea95] Martin S Feather, Steven S Fickas. "Requirements Monitoring in Dynamic Environments". Proceedings of IEEE International Conference on Requirements Engineering, 1995

[Fea97] Feather, M.S. "FLEA: Formal Language for Expressing Assumptions – Language Description", June 25, 1997

[Fea98] M.S. Feather, S. Fickas, A. Van Lamsweerde and C. Ponsard. "Reconciling System Requirements and Runtime Behaviour". Proceeding ISSWD'98 – 9th International Workshop on Software Specifiacation and Design, Isobe, IEEE CS Press, April 1998.

[Fic02a] Stephen Fickas, Lauire Ehlhardt, McKay Sohlberg, Bonnie Todis. "Personal Requirements Engineering". Technical Report 45-02, Computer Science Department, University of Oregon, USA, 2002

[Fic02b] Stephen Fickas, Tiller Beauchamp, Ny Aina Razermera Mamy. "Monitoring Requirements: A Case Study". Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02), 2002.

[Fic02c] Stephen Fickas, Max Skorodinsky, Martin Feather. "Sleeping at Night: Building and Monitoring Better Models of the Environment". Proceedings of the First Workshop on State of the Art on Automated Software Engineering, June 2002.

[Fic02d] Stephen Fickas and Robert J. Hall. "Self-Healing Open Systems". Proceedings of the First Workshop on Self-Healing Systems, 2002, Charleston, South Carolina.

[Fin01] A. Finkelstein and A. Savigni. "A Framework for Requirements Engineering for Context-Aware Services". In Proc. of 1st International Workshop From Software Requirements to Architectures (STRAW 01), Toronto, Canada, May 2001.

[Fit96] Melvin Fitting , "First-order logic and automated theorem proving (2nd ed.)", Springer Graduate Texts In Computer Science, ISBN:0-387-94593-8, 1996

[Fos06] Howard Foster. "A Rigorous Approach to Engineering Web Service Compositions", PhD Thesis, Department of Computing, Imperial College London, January 2006.

[Ftp80] File Transfer Protocol (FTP), http://www.faqs.org/rfcs/rfc765.html

[Gab80] D.M. Gabbay, A. Pnueli, S. Shelah, J. Stavi. On the Temporal Analysis of Fairness. 7th ACM Symposium on Principles of Programming Languages. Las Vegas (1980) 163-173.

[Geo03] George, V. and Vaughn, R., "Application of Lightweight Formal Methods in Requirement Engineering", CROSSTALK, The Journal of Defense Software Engineering, vol 16, no 1, January 2003, p. 30.

[Gib90] P.B. Gibbons, R.M. Karp, G.L. Miller, D. Soroker, "Subtree isomorphism is in random NC", Discrete Applied Mathematics, Discrete Appl. Math. (Netherlands), vol.29, (no.1), p.35-62, November 1990

[Gnu] "The Gnutella Protocol Specification v0.4",

http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[Gun02] Elsa Gunter and Doron Peled. "Tracing the Executions of Concurrent Programs", In Proceedings of Second International Workshop on Runtime Verification, Copenhagen Denmark, 26 July, 2002

[Hal90] A. Hall. "Seven Myths of Formal Methods". IEEE Software September, 1990

[Har00] Mary Jean Harrold, "Testing: A Roadmap", In Future of Software Engineering, 22nd International Conference on Software Engineering, June 2000.

[Hoa85] C.A.R. Hoare. "Communicating Sequential Processes". Prentice Hall, 1985

[Hol91] G.J. Holzmann, "Design and Validation of Computer Protocols". Englewood Cliffs, N.J.: Prentice Hall, 1991

[Hol97] Gerard J. Holzmann. "The Model Checker Spin", - IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 5, MAY 1997

[Hol99] G. Holzmann and M.H. Smith. "Software model checking". In Proceeding Formal Techniques for Networked and Distributed Systems, Beijing, China, 1999

[Hon06] Yang Hongli, Zhao Xiangpeng, and Qiu Zongyan, "A Formal Model for Web Service Choreography Description Language (WS-CDL)", Tech. Report, 2006.

[How04]Howard Barringer, Allen Goldberg, Klaus Havelund, Koushik Sen, "Rule-Based Runtime Verification", In Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation , volume 2937 of LNCS, pages 44-57, Venice, Italy, January 2004, Springer-Verlag.

[Htt99] Hypertext Transfer Protocol (HTTP), http://www.w3.org/Protocols/rfc2616/rfc2616.html

[Hyp03] "Hyper/J",July 8, 2003, http://www.alphaworks.ibm.com/tech/hyperj

[Ijc04] The IJCAR 2004 Workshop on Empirically Succesful First Order Reasonning.

[Iso96] ISO/IEC 14977:1996, "Information technology -- Syntactic metalanguage -- Extended BNF"

[IPL03] "An Introduction to Software Testing", IPL Information Processing Ltd, white paper, September 2003

[Iws00] IBM Web Service Architecture Team. "Web Services Architecture Overview", IBM Developer Works, September 2000.

[Jac06] Daniel Jackson, " Dependable Software by Design", *Scientific American*, June 2006

[Jav94] Sun Microsystems, Inc. © 1994 – 2006, "The Source For Java Technology". http://java.sun.com/

[Jin02] Li-jie Jin, Vijay Machiraju, Akhil Sahai. "Analysis on Service Level Agreement of Web Services". Software Technology Laboratory, HP Laboratories Palo Alto, HPL-2002-180, June 21, 2002.

[Kan00] Sampath Kannan, Moonjoo Kim, Insup Lee, Oleg Sokolsky, Mahesh Viswanathan, "

Run-time Monitoring and Steering based on Formal Specifications", Workshop on Modeling Software System Structures in a fastly moving scenario, June 2000.

[Kan06] Cem Kaner, "Inefficiency and ineffectiveness of software testing: A key problem in software engineering." White paper prepared for the National Defense Industrial Association's Systems Engineering Workshop on the Top 5 Software Issues, Washington, D.C., August 2006.

[Kel02] A. Keller and H. Ludwig. "TheWSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services". Technical Report RC22456(W0205-171), IBM Research Division, T.J. Watson Research Center, May 2002.

[Kel03] Keller A. and Ludwig H., "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services", Journal of Network and Systems Management, vol. 11, num. 1, 2003, p. 57-81, Plenum Publishing.

[Kim01a] Moonjoo Kim, "Information Extraction for Run-time Formal Analysis". PhD thesis, CIS Department, University of Pennsylvania 2001.

[Kim01b] Moonjoo Kim, Sampath Kannan, Insup Lee, O. Sokolsky, and Mahesh Viswanathan, "Java-MaC: a Runtime Assurance Tool for Java Programs", In Klaus Havelund and Grigore Rosu, editors, Electronic Notes in Theoretical Computer Science, volume 55. Elsevier Science Publishers, 2001.

[Kla04] Klaus Havelund , Grigore Roşu, "An Overview of the Runtime Verification Tool Java PathExplorer", Formal Methods in System Design, v.24 n.2, p.189-215, March 2004

[Kre01] Kreger Heather. "Web Services Conceptual Architecture (WSCA 1.0)", IBM Software Group, IBM Web Services, May 2001.

[Lam80] Lamport, L., "Sometime is Sometimes `Not Never', On the Temporal Logic of Programs", Proceedings of the Seventh ACM Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN (January 1980).

[Lam83] Lamport, L., What Good is Temporal Logic". Information Processing 83:657-668, 1983.

[Lam00] A van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", IEEE Transactions on Software Engineering, vol. 26, 2000, pp. 978-1005

[Lam03] David Lamanna, James Skene and Wolfgang Emmerich. "SLAng: A Language for Defining Service Level Agreements". In Proc. of the 9th IEEE Workshop on Future Trends in Computing Systems, San Juan, Puerto Rico. pp. 100-106. IEEE Computer Society Press. June 2003

[Laz04] A. Lazovik, M. Aiello and M. Papazoglou. Associating Assertions with Business

Processes and Monitoring their Execution. In Proceedings of the 2nd International Conference on Service Oriented Computing, 2004.

[Laz06a] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. Journal on Digital Libraries, Springer, 2006.

[Laz06b] A. Lazovik and M. Aiello. Associating Assertions with Business Processes and Monitoring their Execution. International Journal of Cooperative Information Systems, June 2006

[Lea00] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. "JML: notations and tools supporting detailed design in Java". In OOPSLA 2000 Companion, pages 105–106, 2000.

[Lev93] Nancy Leveson, Clark S. Turner, " An Investigation of the Therac-25 Accidents", *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 18-41

[Lev98] H. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus". Electronic Transactions on Artificial Intelligence, 2(3-4):159-178. 1998

[Log03] http://logging.apache.org/log4j/docs/, September 2003.

[Lud03] H. Ludwig, A. Keller, A. Dan, R.P. King, and R. Franck, "Web Service Level Agreement (WSLA) Language Specification", Version 1.0, IBM Corporation (January 2003), http://www.research.ibm.com/wsla.

[Lud04] H. Ludwig, A. Dan, and R. Kearney, " Cremona: An Architecture and Library for Creation and Monitoring of WS-Agreements" Proceedings of the 2nd International Conference on Service Oriented Computing, November 2004, New York

[Lut01] Robyn R. Lutz and Ines Carmen Mikulski. "Evolution of Safety-Critical Requirements Post-Launch". Proceedings of the 5th International Symposium of on Requirements Engineering, IEEE, Toronto, Canada, August 27–31, 2001.

[Mah04] Mahbub K., Spanoudakis G. "A framework for Requirements Monitoring of Service Based Systems", 2nd International Conference on Service Oriented Computing (ICSOC 2004), pp 84 – 93, November 2004.

[Mah05] Mahbub K, Spanoudakis G. "Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience", IEEE International Conference on Web Services (ICWS'05), pp. 257-265, 2005.

[Mah07] Khaled Mahbub, George Spanoudakis, "Monitoring WS-Agreements: An Event Calculus Based Approach", Springer monograph on Test and Analysis of Web Services (to appear in 2007).

[Man95] Z. Manna and A. Pnueli. "Temporal Verification of Reactive Systems: Safety". Springer, New York, 1995.

[Mcm92] K. L. McMillan. *The SMV system*, November 2000. http://www-2.cs.cmu.edu/modelcheck/smv.html

[Men02] Daniel A. Menasce, "QoS Issues in Web Services", IEEE Internet Computing, November-December 2002.

[Men04] Jan Mendling, Gustaf Neumann and Markus Nüttgens, "A Comparison of XML Interchange Formats for Business Process Modelling", In Proceedings of the Workshop organized by the special interest group "EMISA - GI - Development Methods for Information Systems and their Application", Centre de Recherche Public - Gabriel Lippmann, Luxembourg, October 6-8, 2004.

[Mic04] Microsoft Biztalk Server, April 2004, http://www.microsoft.com/biztalk/downloads/default.mspx

[Mil99] Miller, R, and Shanahan, M. "The Event Calculus in Classical Logic". Linkoping Electronic Articles in Computer and Information Science, 4(16). 1999

[Mil04] Nikola Milanovic and Miroslaw Malek, "Current Solutions for Web Service Composition", IEEE Internet Computing, November-December 2004

[Min05] Miniaoui, S., Wentland Forte, M. "XML Mining: From Trees to Strings?". Second International Conference on Intelligent Computing and Information Systems (ICICIS 2005), Cairo, Egypt, March 5-7. 2005.

[Mys95] MySQL Community Edition, 1995-2006, http://dev.mysql.com/downloads/mysql/5.0.html

[Nad03] A. Nadalin (ed.). "Web Services Policy Assertions Language (WS-PolicyAssertions)".www.ibm.com/developerworks/library/ws-polas/,May 2003.

[Nad05] A. Nadalin (ed.): "Web Services Security Policy Language", http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-securitypolicy.pdf, July 2005

[Nak02a] S. Nakajima. "On Verifying Web Service Flows". In Proceedings of the 2002 Symposium on Applications and the Internet (SAINT'02w), pages 223 – 224, Jan 2002.

[Nak02b] Shin Nakajima. "Model-Checking Verification for Reliable Web Services". 17th ACM Annual Conference on Object-Oriented Programming, Systems, Languages and Applications. November 4–8, 2002 Washington State Convention & Trade Center, Seattle, Washington, USA

[Nar02] S. Narayanan and S. A. McIlraith, "Simulation, Verification and Automated Composition of Web Services," presented at Eleventh International World Wide Web Conference (WWW-11), Honolulu, Hawaii, 2002.

[Nen02] Nentwich C., Capra L., Emmerich W. and Finkelstein A. "xlinkit: a Consistency Checking and Smart Link Generation Service". ACM Transactions on Internet Technology, 2(2), May 2002, pp. 151-185

[Ora04] Oracle BPEL Process Manager, v2.1.1, December 2004,

http://www.oracle.com/technology/products/ias/bpel/index.html

[Ora06] Oracle Database 10G Enterprise Edition, April 2006, http://www.oracle.com/database/index.html

[Ouy05] Ouyang, C., Aalst, W.M.P. van der, Breutel, S., Dumas, M., Hofstede, A.H.M. ter, & Verbeek, H.M.W. (2005). "Formal semantics and analysis of control flow in WS-BPEL", BPM Center Report (Ext. rep. 05-13). Eindhoven: BPM Center.

[Owi82] Susan Owicki and Leslie Lamport, "Proving Liveness Properties of Concurrent Programs"ACM Transactions on Programming Languages and Systems, Volume 4 , Issue 3 (July 1982).

[Owl04a] OWL-S: Semantic Markup for Web Services, W3C Member Submission 22 November 2004, http://www.w3.org/Submission/OWL-S/

[Owl04b] OWL Web Ontology Language Overview, W3C Recommendation 10 February 2004, http://www.w3.org/TR/owl-features/

[Pan99] Jiantao Pan, "Software Testing", Departmental Report, Electrical and Computer Engineering Department, Carnegie Mellon University, Spring 1999

[Par92] David Lorge Parna, "Tabular Representation of Relations", Technical Report, CRL Report No. 260, October, 1992

[Pas05] Paschke A., Bichler, M., "SLA Representation, Management and Enforcement - Combining Event Calculus, Deontic Logic, Horn Logic and Event Condition Action Rules", E-Technology, E-Commerce, E-Service EEE05 Conference, Hong Kong, March 2005.

[Pau93] G. Paul. "Approaches to Abductive Reasoning: an overview", Artificial Intelligence, Kluwer Academic Publishers, 7, 109-152, 1993.

[Pel03a] Chris Paletz "Web Services Orchestration. A review of emerging technologies, tools and standards", Hewllett Packard White Paper, January 2003

[Pel03b] C. Peltz. Web Service Orchestration and Choreography. A look at WSCI and BPEL4WS. WebServices Journal, 03(7), July 2003.

[Pet97] D. K. Peters. "Deriving Real-Time Monitors from System Requirements Documentation" Proceedings of the Third IEEE International symposium on Requirements Engineering (RE'97) Doctoral Consortium, January 1997

[Pet02] Dennis K. Peters and David Lorge Parnas. "Requirements-based Monitors for Real-Time systems", IEEE Transactions on Software Engineering, vol. 28, February 2002,

[Pic02] Giacomo Piccinelli, Anthony Finkelstein and Christian Nentwich. "Web Services Need Consistency". 17th ACM Annual Conference on Object-Oriented Programming, Systems, Languages and Applications. November 4–8, 2002 Washington State Convention & Trade Center, Seattle, Washington, USA

[Pin94] Pinto, J. A., Temporal Reasoning in the Situation Calculus, Ph.D. Thesis, Department

of Computer Science, University of Toronto, 1994

[Ple93] Plexousakis D. "Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases", In Proc. of the 19th Int. Conference on Very Large Data Bases, 1993

[Pra05] M.Prasanna, S.N. Sivanandam, R. Venkatesan and R.Sundarrajan, "A SURVEY ON AUTOMATIC TEST CASE GENERATION", Academic Open Internet Journal, Volume 15, 2005

[Rdf04] RDF/XML Syntax Specification. February 10, 2004. http://www.w3.org/TR/rdf-syntax-grammar/

[Reg02] Regular Expressions and the Java Programming Language http://java.sun.com/developer/technicalArticles/releases/1.4regex/, April 2002.

[Rob02] William N. Robinson. "Monitoring Software Requirements using Instrumented Code". – In the Proceedings of the Hawaii International Conference on Systems Sciences, January 7 –10, 2002, Big Island, Hawaii.

[Rob03a] William N. Robinson. "Monitoring Web Service Requirements", In the proceedings of 12th International Conference on Requirements Engineering, 2003

[Rob03b] William N. Robinson. "Monitoring Web Service Interactions", In the proceedings of Workshop On Requirements Engineering and Open Systems (REOS), September 08, 2003, Monterey CA.

[Sam92] M. Mansouri-Samani and M. Sloaman, "Monitoring Distributed Systems (A Survey)", Imperial College Research Report No. DOC92/23

[Sch04] J. Schlimmer (ed.). "Web Services Policy Framework (WS-Policy Framework)". www.ibm.com/developerworks/library/specification/ws-polfram/, September 2004.

[Sco93] Roger S. Scowen, "Extended BNF — A generic base standard". Software Engineering Standards Symposium 1993

[Sha90] Shanahan, M. P. "*Representing continuous change in the event calculus*". Proceedings of the European Conference on Artificial Intelligence (ECAI-90), 1990.

[Sha99] M. Shanahan. "The event calculus explained", In M. J. Wooldridge and M. Veloso, editors, Articial Intelligence Today, Vol. 1600 of LNCS, pages 409--430. Springer, 1999

[Sha04] C. Sharp (ed.). "Web Services Policy Attachment (WS-PolicyAttachment)". www-128.ibm.com/developerworks/library/specification/ws-polatt/, September 2004.

[Smt82] Simple Mail Transfer Protocol (SMTP), http://www.freesoft.org/CIE/RFC/821/index.htm

[Soa03] Simple Object Access Protocol (SOAP) 1.2. June 24, 2003. http://www.w3.org/TR/soap/

[Sol03] Monika Solanki and Charlie Abela, "The Landscape of Markup Languages for Web

Service Composition", May 2003

[Sow03] John F. Sowa, "Processes and Causality", http://www.jfsowa.com/ontology/causal.htm, 2003.

[Spa04] Spanoudakis G., Mahbub K.: Requirements Monitoring for Service-Based Systems: Towards a framework based on Event Calculus , 19th IEEE International Conference on Automated Software Engineering.

[Spa06] Spanoudakis G. and Mahbub K. "Non Intrusive Monitoring of Service Based Systems", International Journal of Cooperative Information Systems, 15(3): 325-358, 2006.

[Spa07] Spanoudakis G, Kloukinas C, Androutsopoulos K: "Towards Security Monitoring Patterns", 22nd Annual ACM Symposium on Applied Computing, Technical Track on Software Verification, March 2007 (to appear).

[Sta04] Ch. Stahl. and K. Schmidt. "A Petri net semantic for BPEL", In Ekkart Kindler, editor, Proc. of 11th Workshop AWPN. Paderborn University, October 2004.

[Sun03] Sun One Studio 8, May 2003, http://developers.sun.com/prodtech/cc/reference/index.jsp

[Tal05] Steve Ross-Talbot, "Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows", Invited lecture, NETTAB, 5-7 October, 2005, Second University of Naples, Naples, Italy.

[Tho91] Thomas A. Henzinger, Zohar Manna, Amir Pnueli, " Temporal Proof Methodologies for Real-time Systems", Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, 1991

[Tia03] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. "A Concept for QoS Integration in Web Services". 1st Web Services Quality Workshop (WQW 2003), in conjunction with 4th International Conference on Web Information Systems Engineering (WISE 2003), Rome, Italy, December 2003.

[Tia04] M. Tian, A. Gramm, H. Ritter, and J. Schiller, "A Survey of current Approaches towards Specification and Management of Quality of Service for Web Services", PIK 3/04.

[Tis00] Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio. "Architectural reflection realising software architectures via reflective activities". In Volker Gruhn, Wolfgang Emmerich, and Stefan Tai, editors, Engineering Distributed Objects (EDO 2000), LNCS, Berlin, 2000. Springer.

[Tos01] Vladamir Tosic, Bernard Pagurek, Babak Esfandiari, Kruti Patel. "On the Management of Compositions of Web Services". Workshop on Object-Oriented Web Services - OOWS (at OOPSLA 2001).

[Tos02] Vladamir Tosic, Bernard Pagurek, Babak Esfandiari, Kruti Patel, Wei Ma. "Web

Service Offerings Language (WSOL) and Web Service Composition Management (WSCM). 17th ACM Annual Conference on Object-Oriented Programming, Systems, Languages and Applications. November 4–8, 2002 Washington State Convention & Trade Center, Seattle, Washington, USA

[Tra99] Eushiuan Tran, "Verification, Validation, Certification", Departmental Report, Electrical and Computer Engineering Department, Carnegie Mellon University, Spring 1999

[Tsu99] Tsur, D. and Shamir, R. "Faster Subtree Isomorphism", Journal of Algorithms, 33, 1999, 267--280

[Ud_IBM] https://uddi.ibm.com/ubr/registry.html

[Ud_MIC] http://uddi.microsoft.com/default.aspx

[Udd00] "UDDI Technical White Paper" – UDDI.org White Paper, 6 Sept 2000.

[Udd02] "The Evolution of UDDI" – UDDI.org White Paper, 19th Jul 2002.

[Udd03] UDDI Spec Technical Committee Specification. October 14, 2003. http://uddi.org/pubs/uddi_v3.htm

[Uml03] Unified Modelling Language (UML), version 1.5, March 01, 2003, http://www.omg.org/technology/documents/formal/uml.htm

[Urt02] David Urting, and Yolande Berbers, "Runtime Veri¯cation of Timing Constraints", Department of Computer Science, Katholieke Universiteit Leuven, Belgium, Report CW345, July 2002.

[Var98] M. Vardi. "Linear vs branching time: A complexity-theoretic perspective". In Proceedings, 13th Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press, 1998.

[Var01] Vardi, M. Y., "Branching vs. linear time: Final showdown", in T. Margaria and W. Yi, (eds.), Proceedings of the 2001.

[Whi95] Stephanie White and Michael Edwards, "A requirements taxonomy for specifying complex systems", First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95) p. 373

[Whi00] James A. Whittaker, "What Is Software Testing? And Why Is It So Hard?", IEEE SOFTWARE January/February 2000.

[Wsa02] "Web Services Architecture" – W3C Working Draft, 14th Nov 2002, http://www.w3.org/TR/2002/WD-ws-arch-20021114

[Wsa04] "Web Services Architecture" – W3C Working Group Note, 11th February 2004, http://www.w3.org/TR/ws-arch/

[Wsb06] " Web Services Business Process Execution Language" Version 2.0, Public Review Draft, 23th August, 2006, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html

[Wsc02] Web Service Choreography Interface (WSCI) 1.0, August 08, 2002. http://www.w3.org/TR/wsci/

[Wsc05] Web Srevices Choreography Description Language, Version 1.0, W3C Candidate Recommendation, November 9, 2005, http://www.w3.org/TR/ws-cdl-10/

[Wsd04] Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language, August 2004. http://www.w3.org/TR/2004/WD-wsdl20-20040803/

[Wsf01] Frank Leymann. Web Services Flow Language (WSFL) version 1.0, IBM Software Group, May 2001.

[Wsg04] "Web Services Glossary" – W3C Working Group Note, 11 February, 2004. http://www.w3.org/TR/ws-gloss/

[Xla01] Satish Thatte. "XLANG: Web Services for Business Process Design". Microsoft Corporation, 2001.

[Xli02] Xlinkit: A Consistency Checking and Smart Link Generation Service. ACM Transactions on Software Engineering and Methodology, pages 151-185, May 2002

[Xme05] XMethods, http://www.xmethods.net/

[Xml97] The XML Data Model, April 27, 1997, http://www.w3.org/XML/Datamodel.html

[Xml04a] XML Schema Part 0: Primer Second Edition. 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[Xml04b] XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004, http://www.w3.org/TR/xmlschema-2/#datatype

[Xml04c] Extensible Markup Language (XML) 1.0 (Third Edition), 04 February 2004, http://www.w3.org/TR/REC-xml/#sec-documents

[Xpa99] XML Path Language (XPath), Version 1.0, November 1999

[Yus04] Yushi, C., Wah, L.E. and Limbu, D.K., "Web Services Composition - An Overview of Standards". Synthesis Journal, Fifth issue, ITSC publication, (pp 137-150), 2004.

[Zho03] Zhou Chen, Chia Liang-Tien, Bilhanan Silverajan, and Lee Bu-Sung. "UX – An Architecture Providing QoS-Aware and Federated Support for UDDI". In proceeding of the first International Conference on Web Services(ICWS03), June 23 - 26, 2003, Monte Carlo Resort, Las Vegas, Nevada, USA.

Appendix A

A.1 XML Schema to Express EC formula in XML

```
<?xml version="1.0" encoding="UTF-8" ?>
 <xs:schema targetNamespace="http://tempuri.org/ec/formula"
         xmlns="http://tempuri.org/ec/formula"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         elementFormDefault="qualified">

   <!-- define formulas -->

  <xs:element name="formulas" type="formulasType"/>

  <!-- definition of complex types -->

   <xs:complexType name="formulasType">
       <xs:sequence>
         <xs:element name="formula" type="formulaType" minOccurs="1"
          maxOccurs="unbounded"/>
       </xs:sequence>
   </xs:complexType>

   <xs:complexType name="formulaType">
        <xs:sequence>
         <xs:element name="quantification" type="quantificationType"
                        minOccurs="1" maxOccurs="unbounded"/>
          <xs:element name="body" type="bodyHeadType" minOccurs="0"/>
          <xs:element name="head" type="bodyHeadType"/>
        </xs:sequence>
        <xs:attribute name="formulaId" type="xs:string"
                                                  use="required"/>
        <xs:attribute name="forChecking" type="xs:boolean"
                                        default="true"/>
        <xs:attribute name="forDeduction" type="xs:boolean"
                                        default="true"/>
   </xs:complexType>

   <xs:complexType name="bodyHeadType">
       <xs:sequence>
         <xs:choice>
              <xs:element name="predicate" type="predicateType"/>
              <xs:element name="relationalPredicate"
                                 type="relationalPredicateType"/>
         <xs:choice>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="operator" type="logicalOperatorType"/>
            <xs:choice>
              <xs:element name="predicate" type="predicateType"/>
              <xs:element name="timePredicate"
                                       type="timePredicateType"/>
              <xs:element name="relationalPredicate"
                                    type="relationalPredicateType"/>
            </xs:choice>
         </xs:sequence>
        </xs:sequence>
   </xs:complexType>

   <xs:complexType name="predicateType">
```

```
      <xs:choice>
        <xs:element name="happens" type="happensType"/>
        <xs:element name="initiates" type="initiatesType"/>
        <xs:element name="holdsAt" type="holdsAtType"/>
        <xs:element name="initially" type="holdsAtType"/>
        <xs:element name="terminates" type="terminatesType"/>
        <xs:element name="clipped" type="clippedType"/>
      <xs:element name="declipped" type="declippedType"/>
      </xs:choice>
      <xs:attribute name="negated" type="xs:boolean"
                                              default="false"/>
      <xs:attribute name="unconstrained" type="xs:boolean"
                                              default="false"/>
</xs:complexType>

<xs:complexType name="timePredicateType">
   <xs:choice>
      <xs:element name="timeEqualTo" type="TimeRelation"/>
      <xs:element name="timeNotEqualTo" type="TimeRelation"/>
      <xs:element name="timeLessThan" type="TimeRelation"/>
      <xs:element name="timeGreaterThan" type="TimeRelation"/>
      <xs:element name="timeLessThanEqualTo" type="TimeRelation"/>
      <xs:element name="timeGreaterThanEqualTo"
                                              type="TimeRelation"/>
   </xs:choice>
</xs:complexType>

<xs:complexType name="holdsAtType">
   <xs:sequence>
      <xs:element name="valueOf" type="fluentType"/>
      <xs:element name="timeVar" type="timeVariableType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="initiatesType">
   <xs:sequence>
      <xs:choice>
          <xs:element name="ir_term" type="irTermType"/>
          <xs:element name="rc_term" type="rcTermType"/>
          <xs:element name="as_term" type="asTermType"/>
      </xs:choice>
      <xs:element name="valueOf" type="fluentType"/>
      <xs:element name="timeVar" type="timeVariableType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="happensType">
   <xs:sequence>
      <xs:choice>
          <xs:element name="ic_term" type="icTermType"/>
          <xs:element name="ir_term" type="irTermType"/>
          <xs:element name="rc_term" type="rcTermType"/>
          <xs:element name="re_term" type="reTermType"/>
          <xs:element name="as_term" type="asTermType"/>
      </xs:choice>
      <xs:element name="timeVar" type="timeVariableType"/>
      <xs:element name="fromTime" type="TimeExpression"/>
      <xs:element name="toTime" type="TimeExpression"/>
   </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="clippedType">
   <xs:sequence>
     <xs:element name="timeVar1" type="timeVariableType"/>
     <xs:element name="valueOf" type="fluentType"/>
     <xs:element name="timeVar2" type="timeVariableType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="declippedType">
   <xs:sequence>
    <xs:element name="timeVar1" type="timeVariableType"/>
     <xs:element name="valueOf" type="fluentType"/>
     <xs:element name="timeVar2" type="timeVariableType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="terminatesType">
   <xs:sequence>
     <xs:choice>
         <xs:element name="ir_term" type="irTermType"/>
         <xs:element name="rc_term" type="rcTermType"/>
         <xs:element name="as_term" type="asTermType"/>
     </xs:choice>
     <xs:element name="valueOf" type="fluentType"/>
     <xs:element name="timeVar" type="timeVariableType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="fluentType">
   <xs:sequence>
      <xs:element name="target">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="variable" type="variableType"/>
           </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="source">
         <xs:complexType>
           <xs:choice>
             <xs:element name="variable" type="variableType"/>
             <xs:element name="operationCall"
                                  type="operationCallType"/>
           </xs:choice>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="quantificationType">
   <xs:sequence>
     <xs:element name="quantifier">
       <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:pattern value="forall|existential"/>
         </xs:restriction>
       </xs:simpleType>
     </xs:element>
    <xs:choice >
     <xs:element name="regularVariable" type="variableType"/>
    <xs:element name="timeVariable" type="timeVariableType"/>
```

```
        </xs:choice>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="icTermType">
      <xs:sequence>
        <xs:element name="operationName" type="xs:string"/>
        <xs:element name="partnerName" type="xs:string"/>
        <xs:element name="id" type="xs:string"/>
        <xs:element name="variable" type="variableType" minOccurs="0"
                                              maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="irTermType">
      <xs:sequence>
        <xs:element name="operationName" type="xs:string"/>
        <xs:element name="partnerName" type="xs:string"/>
        <xs:element name="id" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="rcTermType">
      <xs:sequence>
        <xs:element name="operationName" type="xs:string"/>
        <xs:element name="partnerName" type="xs:string"/>
        <xs:element name="id" type="xs:string"/>
      </xs:sequence>
     </xs:complexType>

    <xs:complexType name="reTermType">
      <xs:sequence>
        <xs:element name="operationName" type="xs:string"/>
        <xs:element name="partnerName" type="xs:string"/>
        <xs:element name="id" type="xs:string"/>
        <xs:element name="variable" type="variableType" minOccurs="0"
                                              maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="asTermType">
      <xs:sequence>
        <xs:element name="operationName" type="xs:string"/>
        <xs:element name="id" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="variableType">
      <xs:sequence>
        <xs:element name="varName" type="xs:string"/>
        <xs:choice>
          <xs:sequence>
            <xs:element name="varType" type="xs:string"/>
            <xs:element name="value" type="xs:string"
                                                  minOccurs="0"/>
          </xs:sequence>
          <xs:element name="array" type="arrayType"/>
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="persistent" type="xs:boolean"
default="false"/>
```

```
      <xs:attribute name="forMatching" type="xs:boolean"
default="true"/>
   </xs:complexType>

   <xs:complexType name="timeVariableType">
      <xs:sequence>
        <xs:element name="varName" type="xs:string"/>
        <xs:element name="varType" type="xs:string"
                                          fixed="TimeVariable"/>
        <xs:element name="value" type="xs:string" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>

   <xs:simpleType name="logicalOperatorType">
       <xs:restriction base="xs:string">
         <xs:pattern value="and|or"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:complexType name="TimeExpression">
      <xs:sequence>
        <xs:element name="time" type="timeVariableType"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
           <xs:choice>
             <xs:element name="plusTime" type="timeVariableType"/>
             <xs:element name="minusTime" type="timeVariableType"/>
             <xs:element name="plus" type="xs:decimal"/>
             <xs:element name="minus" type="xs:decimal"/>
           </xs:choice>
        </xs:sequence>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="TimeRelation">
      <xs:sequence>
         <xs:element name="timeVar1" type="TimeExpression"/>
         <xs:element name="timeVar2" type="TimeExpression"/>
      </xs:sequence>
   </xs:complexType>


   <xs:complexType name="varRelationType">
      <xs:sequence>
         <xs:element name="operand1" type="operandType"/>
         <xs:element name="operand2" type="operandType"/>
      </xs:sequence>
   </xs:complexType>

   <xs:complexType name="relationalPredicateType">
     <xs:sequence>
        <xs:choice>
          <xs:element name="equalTo" type="varRelationType"/>
          <xs:element name="notEqualTo" type="varRelationType"/>
          <xs:element name="lessThan" type="varRelationType"/>
          <xs:element name="greaterThan" type="varRelationType"/>
          <xs:element name="lessThanEqualTo" type="varRelationType"/>
          <xs:element name="greaterThanEqualTo"
                                          type="varRelationType"/>
        </xs:choice>
        <xs:element name="timeVar" type="timeVariableType"/>
     </xs:sequence>
```

```
    </xs:complexType>

    <xs:complexType name="operandType">
         <xs:choice>
           <xs:element name="operationCall" type="operationCallType"/>
           <xs:element name="variable" type="variableType"/>
           <xs:element name="constant" type="constantType"/>
         </xs:choice>
    </xs:complexType>

    <xs:complexType name="operationCallType">
       <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="partner" type="xs:string" minOccurs="0"/>
          <xs:element name="variable" type="variableType"
                                 minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
    </xs:complexType>

    <xs:complexType name="constantType">
       <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="value" type="xs:string"/>
       </xs:sequence>
    </xs:complexType>

    <xs:complexType name="arrayType">
       <xs:sequence>
          <xs:element name="type" type="xs:string"/>
          <xs:element name="index" type="xs:string" minOccurs="0"/>
          <xs:element name="value" type="arrayValueType" minOccurs="0"
                                                 maxOccurs="unbounded"/>
       </xs:sequence>
    </xs:complexType>

    <xs:complexType name="arrayValueType">
       <xs:sequence>
          <xs:element name="indexValue" type="xs:string"/>
          <xs:element name="cellValue" type="xs:string"/>
       </xs:sequence>
    </xs:complexType>

</xs:schema>
```

Appendix B

B.1 XML Schema to Express Monitoring Policy in XML

```
<?xml version="1.0" encoding="UTF-8" ?>
 <xs:schema targetNamespace="http://tempuri.org/ec/policy"
         xmlns="http://tempuri.org/ec/policy"
         xmlns:fns="http://tempuri.org/ec/formula"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         elementFormDefault="qualified">

   <!-- define policy -->

   <xs:element name="policy" type="policyType"/>

   <!-- definition of complex and simple types -->

   <xs:complexType name="policyType">
        <xs:sequence>
          <xs:element name="processSpecification"
                              type="processSpecificationType"/>
          <xs:element name="formulas" type="fns:formulasType"
                                              minOccurs="0"/>
          <xs:element name="monitoringMode"
                                    type="monitoringModeType"/>
          <xs:element name="pollingInterval" type="xs:long"/>
          <xs:element name="eventSource" type="eventSourceType"/>
        </xs:sequence>
   </xs:complexType>

   <xs:complexType name="processSpecificationType">
       <xs:sequence>
           <xs:element name="bpelFile" type="xs:string"/>
           <xs:element name="wsdlFiles" type="wsdlFilesType"/>
       </xs:sequence>
   </xs:complexType>

   <xs:complexType name="wsdlFilesType">
       <xs:sequence>
           <xs:element name="wsdlFile" type="xs:string" minOccurs="0"
                                        maxOccurs="unbounded"/>
       </xs:sequence>
   </xs:complexType>

   <xs:simpleType name="monitoringModeType">
       <xs:restriction base="xs:string">
         <xs:pattern value="recorded|mixed"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:complexType name="eventSourceType">
       <xs:sequence>
           <xs:element name="bpelEngineName" type="xs:string"/>
           <xs:element name="ipAddress" type="xs:string"/>
           <xs:element name="port" type="xs:int"/>
       </xs:sequence>
   </xs:complexType>
</xs:schema>
```

B.2 XML Schema to Express Simulator Configuration in XML

```
<?xml version="1.0" encoding="UTF-8" ?>
 <xs:schema targetNamespace="http://tempuri.org/ec/simConfig"
         xmlns="http://tempuri.org/ec/simConfig"
         xmlns:fns="http://tempuri.org/ec/formula"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         elementFormDefault="qualified">

   <!-- define simulation configuration -->

   <xs:element name="simConfig" type="simConfigType"/>

   <!-- definition of complex and simple types -->

   <xs:complexType name="simConfigType">
        <xs:sequence>
          <xs:element name="processSpecification"
                          type="fns:processSpecificationType"/>
          <xs:element name="domains" type="domainsType"/>
          <xs:element name="execPathDistribution"
                                type="distributionType"/>
          <xs:element name="unconstrainedDistribution"
                                    type="distributionType"/>
        </xs:sequence>
   </xs:complexType>

   <xs:complexType name="domainsType">
       <xs:sequence>
           <xs:element name="domain" type="domainType" minOccurs="1"
                                                 maxOccurs="unbounded"/>
       </xs:sequence>
   </xs:complexType>

   <xs:complexType name="domainType">
       <xs:sequence>
           <xs:element name="varName" type="xs:string"/>
           <xs:element name="varType" type="xs:string"/>
           <xs:element name="size" type="xs:int"/>
       </xs:sequence>
   </xs:complexType>

   <xs:complexType name="distributionType">
       <xs:sequence>
           <xs:element name="mean" type="xs:float"/>
           <xs:element name="variance" type="xs:float"/>
       </xs:sequence>
   </xs:complexType>

</xs:schema>
```

Appendix C

C.1 XML Schema to Express Templates in XML

```
<?xml version="1.0" encoding="UTF-8" ?>
 <xs:schema targetNamespace="http://tempuri.org/ec/template"
         xmlns="http://tempuri.org/ec/template"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         elementFormDefault="qualified">


   <!-- define templates -->

  <xs:element name="templates" type="templatesType"/>

   <!-- definition of complex types -->

   <xs:complexType name="templatesType">
       <xs:sequence>
         <xs:element name="template" type="templateType"
                          minOccurs="1" maxOccurs="unbounded"/>
       </xs:sequence>
   </xs:complexType>

   <xs:complexType name="templateType">
        <xs:sequence>
          <xs:element name="dependants" type="dependantsType"/>
          <xs:element name="varBindings" type="varBindingsType"/>
          <xs:element name="status">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:pattern
                       value="UD|I_R_B|I_E_B|U_B|P_I_E_B|P_U_B"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:element>
          <xs:element name="body" type="predicatesType"/>
          <xs:element name="head" type="predicatesType"/>
        </xs:sequence>
        <xs:element name="decisionTime" type="xs:string"/>
        <xs:element name="copyCounter" type="xs:string"/>
        <xs:attribute name="formulaId" type="xs:string"
                                                 use="required"/>
        <xs:attribute name="type" type="xs:string" use="required"/>
        <xs:attribute name="active" type="xs:boolean"
                                               default="false"/>
   </xs:complexType>

   <xs:complexType name="dependantsType">
        <xs:sequence>
          <xs:element name="dependant" type="dependantType"
                           minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
   </xs:complexType>

   <xs:complexType name="dependantType">
        <xs:sequence>
          <xs:element name="targetId" type="xs:string"/>
          <xs:element name="type">
```

```
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:pattern value="abductive|deductive"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="signature" type="signatureType"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="varBindingsType">
      <xs:sequence>
        <xs:element name="variable" type="variableType"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="predicatesType">
      <xs:sequence>
        <xs:element name="predicate" type="predicateType"
                          minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
</xs:complexType>

<xs:complexType name="predicateType">
      <xs:sequence>
        <xs:choice>
           <xs:element name="signature" type="signatureType"/>
           <xs:element name="relationalPredicate"
                                type="relationalPredicateType"/>
        </xs:choice>
        <xs:element name="timeVarQuantifier"
                                        type="quantifierType"/>
        <xs:element name="timeVar" type="timeVariableType"/>
        <xs:element name="fromTime" type="TimeExpression"/>
        <xs:element name="toTime" type="TimeExpression"/>
        <xs:element name="truthValue" type="truthValueType"/>
        <xs:element name="source" type="sourceType"/>
        <xs:element name="ID" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="pHappens" type="xs:boolean"
                                              default="false"/>
      <xs:attribute name="negated" type="xs:boolean"
                                              default="false"/>
</xs:complexType>


<xs:simpleType name="truthValueType">
    <xs:restriction base="xs:string">
      <xs:pattern value="True|False|Un"/>
   </xs:restriction>
</xs:simpleType>

<xs:simpleType name="sourceType">
    <xs:restriction base="xs:string">
      <xs:pattern value="UN|RE|DE|NF"/>
   </xs:restriction>
</xs:simpleType>

<xs:simpleType name="quantifierType">
    <xs:restriction base="xs:string">
      <xs:pattern value="forall|existential"/>
```

```
   </xs:restriction>
</xs:simpleType>

<xs:complexType name="signatureType">
     <xs:choice>
       <xs:element name="happens" type="happensType"/>
       <xs:element name="initiates" type="initiatesType"/>
       <xs:element name="holdsAt" type="holdsAtType"/>
     </xs:choice>
</xs:complexType>

<xs:complexType name="holdsAtType">
   <xs:sequence>
     <xs:element name="valueOf" type="fluentType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="initiatesType">
   <xs:sequence>
     <xs:choice>
         <xs:element name="ir_term" type="irTermType"/>
         <xs:element name="rc_term" type="rcTermType"/>
         <xs:element name="as_term" type="asTermType"/>
     </xs:choice>
     <xs:element name="valueOf" type="fluentType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="happensType">
   <xs:sequence>
     <xs:choice>
         <xs:element name="ic_term" type="icTermType"/>
         <xs:element name="ir_term" type="irTermType"/>
         <xs:element name="rc_term" type="rcTermType"/>
         <xs:element name="re_term" type="reTermType"/>
         <xs:element name="as_term" type="asTermType"/>
     </xs:choice>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="relationalPredicateType">
  <xs:sequence>
     <xs:choice>
       <xs:element name="equalTo" type="varRelationType"/>
       <xs:element name="notEqualTo" type="varRelationType"/>
       <xs:element name="lessThan" type="varRelationType"/>
       <xs:element name="greaterThan" type="varRelationType"/>
       <xs:element name="lessThanEqualTo" type="varRelationType"/>
       <xs:element name="greaterThanEqualTo"
                                          type="varRelationType"/>
     </xs:choice>
     <xs:element name="timeVar" type="timeVariableType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fluentType">
   <xs:sequence>
      <xs:element name="target">
         <xs:complexType>
           <xs:sequence>
             <xs:element name="variable" type="variableType"/>
```

```
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="source">
          <xs:complexType>
            <xs:choice>
              <xs:element name="variable" type="variableType"/>
              <xs:element name="operationCall"
                                        type="operationCallType"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="TimeExpression">
    <xs:sequence>
      <xs:element name="time" type="timeVariableType"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
         <xs:choice>
           <xs:element name="plusTime" type="timeVariableType"/>
           <xs:element name="minusTime" type="timeVariableType"/>
           <xs:element name="plus" type="xs:decimal"/>
           <xs:element name="minus" type="xs:decimal"/>
         </xs:choice>
      </xs:sequence>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="icTermType">
  <xs:sequence>
      <xs:element name="operationName" type="xs:string"/>
      <xs:element name="partnerName" type="xs:string"/>
      <xs:element name="variable" type="variableType" minOccurs="0"
                                          maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="irTermType">
    <xs:sequence>
      <xs:element name="operationName" type="xs:string"/>
      <xs:element name="partnerName" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="rcTermType">
    <xs:sequence>
      <xs:element name="operationName" type="xs:string"/>
      <xs:element name="partnerName" type="xs:string"/>
    </xs:sequence>
 </xs:complexType>

<xs:complexType name="reTermType">
    <xs:sequence>
      <xs:element name="operationName" type="xs:string"/>
      <xs:element name="partnerName" type="xs:string"/>
      <xs:element name="variable" type="variableType" minOccurs="0"
                                        maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="asTermType">
   <xs:sequence>
     <xs:element name="operationName" type="xs:string"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="variableType">
   <xs:sequence>
     <xs:element name="varName" type="xs:string"/>
     <xs:choice>
        <xs:sequence>
           <xs:element name="varType" type="xs:string"/>
           <xs:element name="value" type="xs:string"
                                          minOccurs="0"/>
        </xs:sequence>
        <xs:element name="array" type="arrayType"/>
     </xs:choice>
   </xs:sequence>
   <xs:attribute name="persistent" type="xs:boolean"
                                       default="false"/>
   <xs:attribute name="forMatching" type="xs:boolean"
                                          default="true"/>
</xs:complexType>

<xs:complexType name="timeVariableType">
   <xs:sequence>
     <xs:element name="varName" type="xs:string"/>
     <xs:element name="varType" type="xs:string"
                                       fixed="TimeVariable"/>
     <xs:element name="value" type="xs:string" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="varRelationType">
   <xs:sequence>
      <xs:element name="operand1" type="operandType"/>
      <xs:element name="operand2" type="operandType"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="relationalPredicateType">
  <xs:sequence>
     <xs:choice>
       <xs:element name="equalTo" type="varRelationType"/>
       <xs:element name="notEqualTo" type="varRelationType"/>
       <xs:element name="lessThan" type="varRelationType"/>
       <xs:element name="greaterThan" type="varRelationType"/>
       <xs:element name="lessThanEqualTo" type="varRelationType"/>
       <xs:element name="greaterThanEqualTo"
                                          type="varRelationType"/>
     </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="operandType">
     <xs:choice>
       <xs:element name="operationCall" type="operationCallType"/>
       <xs:element name="variable" type="variableType"/>
       <xs:element name="constant" type="constantType"/>
     </xs:choice>
</xs:complexType>
```

```
    <xs:complexType name="operationCallType">
       <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="partner" type="xs:string" minOccurs="0"/>
          <xs:element name="variable" type="variableType"
                                  minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
    </xs:complexType>

    <xs:complexType name="constantType">
       <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="value" type="xs:string"/>
       </xs:sequence>
    </xs:complexType>

    <xs:complexType name="arrayType">
       <xs:sequence>
          <xs:element name="type" type="xs:string"/>
          <xs:element name="index" type="xs:string" minOccurs="0"/>
          <xs:element name="value" type="arrayValueType" minOccurs="0"
                                              maxOccurs="unbounded"/>
       </xs:sequence>
    </xs:complexType>

    <xs:complexType name="arrayValueType">
       <xs:sequence>
          <xs:element name="indexValue" type="xs:string"/>
          <xs:element name="cellValue" type="xs:string"/>
       </xs:sequence>
    </xs:complexType>

</xs:schema>
```

Appendix D

Specification of the First Case Study

D.1 BPEL Specification of the CRS

```
<process name="carServiceProcess"
     targetNamespace="http://carservice.org/carserviceprocessing"
     suppressJoinFailure="yes"
     xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
     xmlns:sns="http://carservice.org/wsdl/OnlineRenter"
     xmlns:crns="http://tempuri.org/services/CarReg"
     xmlns:csns="http://tempuri.org/services/CustomerReg">

   <variables>
     <variable name="authRes"
                        messageType="csns:authenticateResponse"/>
     <variable name="authReq"
                         messageType="csns:authenticateRequest"/>
     <variable name="isAvailReq"
                         messageType="crns:isAvailableRequest"/>
     <variable name="isAvailRes"
                          messageType="crns:isAvailableResponse"/>
     <variable name="depReq" messageType="sns:departRequest"/>
     <variable name="depRes" messageType="sns:departResponse"/>
     <variable name="entReq" messageType="sns:enterRequest"/>
     <variable name="entRes" messageType="sns:enterResponse"/>
     <variable name="retKeyReq" messageType="sns:retKeyRequest"/>
     <variable name="retKeyRes" messageType="sns:retKeyResponse"/>
     <variable name="mkUnAvailReq"
                         messageType="crns:makeUnAvailableRequest"/>
     <variable name="mkUnAvailRes"
                        messageType="crns:makeUnAvailableResponse"/>
     <variable name="mkAvailReq"
                          messageType="crns:makeAvailableRequest"/>
     <variable name="mkAvailRes"
                         messageType="crns:makeAvailableResponse"/>
     <variable name="bpelReq" messageType="sns:request"/>
     <variable name="bpelRes" messageType="sns:response"/>
   </variables>

   <partners>
      <partner name="CRS" serviceLinkType="sns:CarRenterLT"
                                          myRole="RentManager"/>
      <partner name="CMS" serviceLinkType="sns:CustomerManagerLT"
                                 partnerRole="CustomerManager"/>
      <partner name="CRMS" serviceLinkType="sns:CarManagerLT"
                                       partnerRole="CarManager"/>
   </partners>

   <correlationSets>
        <correlationSet name="locInfo" properties="sns:locs"/>
        <correlationSet name="carInfo" properties="sns:car_id"/>
        <correlationSet name="custInfo" properties="sns:cust_id"/>
        <correlationSet name="locAndCarInfo" properties="sns:locs
                                                  sns:car_id"/>
   </correlationSets>
```

```
<flow>
   <links>
         <link name="receive-to-auth"/>
         <link name="auth-to-check"/>
         <link name="check-to-car"/>
         <link name="car-to-reply"/>
         <link name="check-to-noCar"/>
         <link name="noCar-to-reply"/>
         <link name="auth-to-no"/>
         <link name="no-to-reply"/>
         <link name="enter-to-retKey"/>
         <link name="release-to-depNotDep"/>
   </links>

   <receive name="receive1" partner="CRS" portType="sns:CarRenter"
         operation="receiveRequest" variable="bpelReq"
                                              createInstance="yes">
         <source name="rToa" linkName="receive-to-auth"/>
         <correlations>
               <correlation set="locInfo" initiate="yes"/>
               <correlation set="custInfo" initiate="yes"/>
               <correlation set="locAndCarInfo" initiate="yes"/>
         </correlations>
   </receive>
   <assign name="assign1">
         <target linkName="auth-to-no"/>
         <source linkName="no-to-reply"/>
         <copy>
               <from expression="'Customer Not Authenticated'"/>
               <to variable="bpelRes" part="carId"/>
         </copy>
   </assign>

   <sequence>
         <target linkName="auth-to-check"/>
         <assign name="assign2">
               <copy>
                     <from variable="bpelReq" part="loc"/>
                     <to variable="isAvailReq" part="loc"/>
               </copy>
         </assign>
         <invoke name="invokeIsAvail" partner="CRMS"
                portType="crns:CarReg" operation="isAvailable"
                inputVariable="isAvailReq"
                outputVariable="isAvailRes">

                <source linkName="check-to-noCar"
                      transitionCondition="bpws:getVariableData
                         ('isAvailRes','carId') = 'null'"/>
               <source linkName="check-to-car"
                      transitionCondition="bpws:getVariableData
                        ('isAvailRes' , 'carId') != 'null'"/>

               <correlations>
                    <correlation set="carInfo" initiate="yes"
                                                 pattern="in"/>
                     <correlation set="locAndCarInfo"
                                  initiate="yes" pattern="in"/>
               </correlations>
         </invoke>
   </sequence>
```

```
<assign name="assign3">
      <target linkName="check-to-car"/>
      <source linkName="car-to-reply"/>
      <copy>
            <from variable="isAvailRes" part="carId"/>
            <to variable="bpelRes" part="carId"/>
      </copy>
</assign>
<assign name="assign4">
      <target linkName="check-to-noCar"/>
      <source linkName="noCar-to-reply"/>
      <copy>
            <from expression="'Car Not Available'"/>
            <to variable="bpelRes" part="carId"/>
      </copy>
</assign>

<sequence>
      <target linkName="receive-to-auth"/>
      <assign name="assign5">
            <copy>
                  <from variable="bpelReq" part="custId"/>
                  <to variable="authReq" part="custId"/>
            </copy>
      </assign>
      <invoke name="invokeAuth" partner="CMS"
           portType="csns:CustomerReg" operation="authenticate"
           inputVariable="authReq" outputVariable="authRes">

           <source name="aToc" linkName="auth-to-check"
              transitionCondition="(bpws:getVariableData
              ('authRes' , 'authenticateReturn') = true())"/>
            <source name="aTon" linkName="auth-to-no"
               transitionCondition="bpws:getVariableData
               ('authRes' , 'authenticateReturn') = false()"/>
      </invoke>
</sequence>

<reply name="reply" partner="CRS" portType="sns:CarRenter"
            operation="receiveRequest" variable="bpelRes">
      <target linkName="car-to-reply"/>
      <source linkName="release-to-depNotDep"/>
      <correlations>
            <correlation set="carInfo"/>
      </correlations>
</reply>

<reply name="reply" partner="CRS" portType="sns:CarRenter"
            operation="receiveRequest" variable="bpelRes">
      <target linkName="noCar-to-reply"/>
      <target linkName="no-to-reply"/>
      <correlations>
            <correlation set="carInfo"/>
      </correlations>
</reply>

<sequence>
   <receive name="receive2" partner="CRS"
         portType="sns:CarRenter" operation="enter"
                                          variable="entReq">
```

```
      <correlations>
            <correlation set="locInfo" initiate="yes"/>
            <correlation set="carInfo" initiate="no"/>
      </correlations>
      </receive>
      <reply partner="CRS" portType="sns:CarRenter"
                  operation="enter" variable="entRes"/>
         <source linkName="enter-to-retKey"/>
</sequence>
<pick>
  <target linkName="enter-to-retKey"/>
  <onMessage partner="CRS" portType="sns:CarRenter"
               operation="returnKey" variable="retKeyReq">
    <correlations>
            <correlation set="locInfo"/>
            <correlation set="carInfo"/>
    </correlations>
    <sequence>
         <assign name="assign6">
            <copy>
                  <from variable="retKeyReq" part="carId"/>
                  <to variable="mkAvailReq" part="carId"/>
            </copy>
          </assign>
          <assign name="assign7">
             <copy>
                  <from variable="retKeyReq" part="loc"/>
                  <to variable="mkAvailReq" part="loc"/>
             </copy>
          </assign>
          <invoke name="mkAvail1" partner="CRMS"
             portType="crns:CarReg" operation="makeAvailable"
             inputVariable="mkAvailReq"
                                 outputVariable="mkAvailRes">

            <correlations>
                  <correlation set="locInfo" pattern="out"/>
                  <correlation set="carInfo" pattern="out"/>
            </correlations>
          </invoke>
          <reply partner="CRS" portType="sns:CarRenter"
                operation="returnKey" variable="retKeyRes"/>
     </sequence>
   </onMessage>
   <onAlarm for="'PT90S'">
          <empty/>
   </onAlarm>
</pick>
<pick>
   <target linkName="release-to-depNotDep"/>
   <onMessage partner="CRS" portType="sns:CarRenter"
                  operation="depart" variable="depReq">
      <correlations>
            <correlation set="locAndCarInfo" initiate="no"/>
      </correlations>
      <sequence>
         <assign name="assign8">
           <copy>
                <from variable="depReq" part="carId"/>
                <to variable="mkUnAvailReq" part="carId"/>
           </copy>
```

```
            </assign>
            <assign name="assign9">
               <copy>
                  <from variable="depReq" part="loc"/>
                  <to variable="mkUnAvailReq" part="loc"/>
               </copy>
            </assign>
            <assign name="assign10">
               <copy>
                  <from variable="bpelReq" part="custId"/>
                  <to variable="mkUnAvailReq" part="custId"/>
               </copy>
            </assign>
            <invoke name="mkUnAvail" partner="CRMS"
                portType="crns:CarReg" operation="makeUnAvailable"
                inputVariable="mkUnAvailReq"
                              outputVariable="mkUnAvailRes">

               <correlations>
                     <correlation set="locInfo" pattern="out"/>
                     <correlation set="carInfo" pattern="out"/>
                     <correlation set="custInfo" pattern="out"/>
               </correlations>
            </invoke>
            <reply partner="CRS" portType="sns:CarRenter"
                           operation="depart" variable="depRes"/>
         </sequence>
      </onMessage>
      <onAlarm for="'PT30S'">
          <sequence>
             <assign name="assign11">
                <copy>
                   <from variable="isAvailRes" part="carId"/>
                   <to variable="mkAvailReq" part="carId"/>
                </copy>
             </assign>
             <assign name="assign12">
                <copy>
                   <from variable="bpelReq" part="loc"/>
                   <to variable="mkAvailReq" part="loc"/>
                </copy>
             </assign>
             <invoke name="mkAvail2" partner="CRMS"
                portType="crns:CarReg" operation="makeAvailable"
                inputVariable="mkAvailReq"
                               outputVariable="mkAvailRes">

                <correlations>
                   <correlation set="locInfo" pattern="out"/>
                   <correlation set="carInfo" pattern="out"/>
                </correlations>
                </invoke>
          </sequence>
          </onAlarm>
       </pick>
  </flow>
</process>
```

D.2 WSDL Specification of the CRS

```
<definitions
   targetNamespace="http://carservice.org/wsdl/OnlineRenter"
   xmlns:tns="http://carservice.org/wsdl/OnlineRenter"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:slnk="http://schemas.xmlsoap.org/ws/2003/03/service-link/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:crns="http://tempuri.org/services/CarReg"
   xmlns:csns="http://tempuri.org/services/CustomerReg">


  <import namespace="http://tempuri.org/services/CustomerReg"
      location=
      "http://138.40.91.72:8080/wstk/CustomerReg/CustomerReg.wsdl"/>

  <import namespace="http://tempuri.org/services/CarReg"
     location="http://138.40.91.72:8080/wstk/CarReg/CarReg.wsdl"/>


  <message name="request">
      <part name="custId" type="xsd:string"/>
      <part name="loc" type="xsd:string"/>
  </message>

  <message name="response">
      <part name="carId" type="xsd:string"/>
  </message>

  <message name="departRequest">
      <part name="carId" type="xsd:string"/>
      <part name="loc" type="xsd:string"/>
  </message>

  <message name="departResponse">

  </message>

  <message name="enterRequest">
      <part name="carId" type="xsd:string"/>
      <part name="loc" type="xsd:string"/>
  </message>

  <message name="enterResponse">

  </message>

  <message name="retKeyRequest">
      <part name="carId" type="xsd:string"/>
      <part name="loc" type="xsd:string"/>
  </message>

  <message name="retKeyResponse">

  </message>

  <portType name="CarRenter">
      <operation name="receiveRequest">
            <input message="tns:request"/>
            <output message="tns:response"/>
```

```
    </operation>
    <operation name="depart">
          <input message="tns:departRequest"/>
          <output message="tns:departResponse"/>
    </operation>
    <operation name="enter">
          <input message="tns:enterRequest"/>
          <output message="tns:enterResponse"/>
    </operation>
    <operation name="returnKey">
          <input message="tns:retKeyRequest"/>
          <output message="tns:retKeyResponse"/>
    </operation>
</portType>

<slnk:serviceLinkType name="CarRenterLT">
    <slnk:role name="RentManager">
          <portType name="tns:CarRenter"/>
    </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="CustomerManagerLT">
  <slnk:role name="CustomerManager">
    <portType name="csns:CustomerReg"/>
  </slnk:role>
</slnk:serviceLinkType>

<slnk:serviceLinkType name="CarManagerLT">
  <slnk:role name="CarManager">
    <portType name="crns:CarReg"/>
  </slnk:role>
</slnk:serviceLinkType>

 <property name="car_id" type="xsd:string"/>
 <property name="locs" type="xsd:string"/>
 <property name="cust_id" type="xsd:string"/>

<propertyAlias propertyName="tns:cust_id"
                     messageType="tns:request"
                     part="custId"
                     query="/custId"/>

<propertyAlias propertyName="tns:cust_id"
                     messageType="csns:authenticateRequest"
                     part="custId"
                     query="/custId"/>

<propertyAlias propertyName="tns:locs"
                     messageType="tns:departRequest"
                     part="loc"
                     query="/loc"/>

 <propertyAlias propertyName="tns:locs"
                     messageType="tns:request"
                     part="loc"
                     query="/loc"/>

 <propertyAlias propertyName="tns:locs"
                     messageType="crns:isAvailableRequest"
                     part="loc"
                     query="/loc"/>
```

```
  <propertyAlias propertyName="tns:car_id"
                       messageType="crns:isAvailableResponse"
                       part="isAvailableReturn"
                       query="/isAvailableReturn"/>

  <propertyAlias propertyName="tns:car_id"
                       messageType="tns:departRequest"
                       part="carId"
                       query="/carId"/>

  <!-- The service name and the TNS represent my service ID QName -->
  <service name="carServiceBP"/>

</definitions>
```

D.3 WSDL Specification of the Car Information System (IS)

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
      targetNamespace="http://tempuri.org/services/CarReg"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl=http://tempuri.org/services/CarReg
      xmlns:intf="http://tempuri.org/services/CarReg"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:message name="makeAvailableResponse">

   </wsdl:message>

   <wsdl:message name="makeUnAvailableRequest">
      <wsdl:part name="carId" type="xsd:string"/>
      <wsdl:part name="custId" type="xsd:string"/>
      <wsdl:part name="loc" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="isAvailableResponse">
      <wsdl:part name="carId" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="makeAvailableRequest">
      <wsdl:part name="carId" type="xsd:string"/>
      <wsdl:part name="loc" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="isAvailableRequest">
      <wsdl:part name="loc" type="xsd:string"/>
   </wsdl:message>

   <wsdl:message name="makeUnAvailableResponse">

   </wsdl:message>

   <wsdl:portType name="CarReg">
      <wsdl:operation name="isAvailable" parameterOrder="loc">
         <wsdl:input message="impl:isAvailableRequest"
```

```
                                name="isAvailableRequest"/>
      <wsdl:output message="impl:isAvailableResponse"
                               name="isAvailableResponse"/>

   </wsdl:operation>

   <wsdl:operation name="makeUnAvailable" parameterOrder="carId
                                                    custId loc">

      <wsdl:input message="impl:makeUnAvailableRequest"
                                  name="makeUnAvailableRequest"/>
      <wsdl:output message="impl:makeUnAvailableResponse"
                                  name="makeUnAvailableResponse"/>
   </wsdl:operation>

   <wsdl:operation name="makeAvailable" parameterOrder="carId
                                                          loc">
      <wsdl:input message="impl:makeAvailableRequest"
                                    name="makeAvailableRequest"/>
      <wsdl:output message="impl:makeAvailableResponse"
                                   name="makeAvailableResponse"/>
   </wsdl:operation>

</wsdl:portType>

<wsdl:binding name="CarRegServiceSoapBinding" type="impl:CarReg">
   <wsdlsoap:binding style="rpc"
               transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="isAvailable">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="isAvailableRequest">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://tempuri.org/services/CarReg"
                                              use="encoded"/>
      </wsdl:input>
      <wsdl:output name="isAvailableResponse">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/CarReg"
                                               use="encoded"/>
      </wsdl:output>
   </wsdl:operation>

   <wsdl:operation name="makeUnAvailable">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="makeUnAvailableRequest">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/CarReg"
                                               use="encoded"/>
      </wsdl:input>
      <wsdl:output name="makeUnAvailableResponse">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/CarReg"
                                              use="encoded"/>
      </wsdl:output>
   </wsdl:operation>

   <wsdl:operation name="makeAvailable">
```

```
            <wsdlsoap:operation soapAction=""/>
            <wsdl:input name="makeAvailableRequest">
               <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://tempuri.org/services/CarReg"
                                                        use="encoded"/>
            </wsdl:input>

            <wsdl:output name="makeAvailableResponse">
               <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://tempuri.org/services/CarReg"
                                                        use="encoded"/>
            </wsdl:output>
         </wsdl:operation>
      </wsdl:binding>

      <wsdl:service name="CarRegService">
         <wsdl:port binding="impl:CarRegServiceSoapBinding"
                                                 name="CarRegService">
            <wsdlsoap:address location=
                "http://138.40.91.72:8080/wstk/services/CarRegService"/>
         </wsdl:port>
      </wsdl:service>
   </wsdl:definitions>
```

Appendix E

Specification of the Second Case Study

E.1 BPEL Specification of the QTP

```
<process name="QuoteTrackerProcess"
      targetNamespace="http://tempuri.org/services/PriceTrackerBpel"
      suppressJoinFailure="yes"
      xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:wsdlns="http://tempuri.org/services/PriceTrackerWsdl"
      xmlns:qns="http://www.themindelectric.com/wsdl/net.xmethods.
            services.stockquote.StockQuote/"
      xmlns:cexns="http://www.xmethods.net/sd/
                              CurrencyExchangeService.wsdl"
      xmlns:calns="http://tempuri.org/services/calc">

  <variables>
    <variable name="quoteRequest"
                    messageType="qns:getQuoteRequest1"/>
    <variable name="quoteResponse"
                    messageType="qns:getQuoteResponse1"/>
    <variable name="countryRequest"
                    messageType="wsdlns:getCountryNameRequest"/>
    <variable name="countryResponse"
                    messageType="wsdlns:getCountryNameResponse"/>
    <variable name="rateRequest" messageType="cexns:getRateRequest"/>
    <variable name="rateResponse"
                    messageType="cexns:getRateResponse"/>
    <variable name="multiplyRequest"
                    messageType="calns:multRequest"/>
    <variable name="multiplyResponse"
                    messageType="calns:multResponse"/>
  </variables>

  <partners>
    <partner name="customer"
              serviceLinkType="wsdlns:QuoteTrackerLinkType"
              myRole="QuoteTracker"/>
    <partner name="QuoteFinder"
              serviceLinkType="wsdlns:QuoteFinderLinkType"
              myRole="QuoteFinder" partnerRole="QuoteFinder"/>
    <partner name="currencyExchanger"
              serviceLinkType="wsdlns:currencyExchangerLinkType"
              partnerRole="currencyExchanger"/>
    <partner name="calculator"
              serviceLinkType="wsdlns:calculatorLinkType"
              partnerRole="calculator"/>
  </partners>

  <correlationSets>
     <correlationSet name="qid" properties="wsdlns:q_id"/>
  </correlationSets>

  <sequence>
     <receive name="receive1" partner="customer"
          portType="qns:net.xmethods.services.
          stockquote.StockQuotePortType" operation="getQuote"
```

```
    variable="quoteRequest" createInstance="yes">

    <correlations>
        <correlation set="qid" initiate="yes"/>
    </correlations>
</receive>
<invoke name="invoke1" partner="QuoteFinder"
   portType="qns:net.xmethods.services.
   stockquote.StockQuotePortType" operation="getQuote"
   inputVariable="quoteRequest" outputVariable="quoteResponse">
</invoke>
<reply name="reply1" partner="customer"
    portType="qns:net.xmethods.services.
    stockquote.StockQuotePortType" operation="getQuote"
    variable="quoteResponse">
    <correlations>
        <correlation set="qid"/>
    </correlations>
</reply>

<pick>
  <onMessage partner="customer"
     portType="wsdlns:QuoteTrackerPortType"
     operation="getCountryName" variable="countryRequest">
     <correlations>
         <correlation set="qid"/>
     </correlations>

     <sequence>
         <assign name="assign1">
           <copy>
              <from variable="countryRequest" part="country"/>
              <to variable="rateRequest" part="country2"/>
           </copy>
         </assign>
         <reply partner="customer"
           portType="wsdlns:QuoteTrackerPortType"
           operation="getCountryName"
           variable="countryResponse"/>
     </sequence>
  </onMessage>
  <onAlarm for="'PT30S'">
     <sequence>
       <assign name="assign3">
         <copy>
            <from expression="'uk'"/>
            <to variable="rateRequest" part="country2"/>
         </copy>
       </assign>
     </sequence>
  </onAlarm>
</pick>

<assign name="assign5">
   <copy>
      <from expression="'usa'"/>
      <to variable="rateRequest" part="country1"/>
   </copy>
</assign>
<invoke name="invoke2" partner="currencyExchanger"
   portType="cexns:CurrencyExchangePortType"
```

```
        operation="getRate" inputVariable="rateRequest"
        outputVariable="rateResponse">
      </invoke>
      <assign name="assign6">
        <copy>
          <from variable="rateResponse" part="Result"/>
          <to variable="multiplyRequest" part="in0"/>
        </copy>
      </assign>
      <assign name="assign7">
        <copy>
          <from variable="quoteResponse" part="Result"/>
          <to variable="multiplyRequest" part="in1"/>
        </copy>
      </assign>
      <invoke name="invoke3" partner="calculator"
            portType="calns:Calculator" operation="mult"
            inputVariable="multiplyRequest"
            outputVariable="multiplyResponse">
      </invoke>
    </sequence>
</process>
```

E.2 WSDL Specification of the QTP

```
<definitions
   targetNamespace="http://tempuri.org/services/PriceTrackerWsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:slnk="http://schemas.xmlsoap.org/ws/2003/03/service-link/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://tempuri.org/services/PriceTrackerWsdl"
   xmlns:qns="http://www.themindelectric.com/wsdl/net.xmethods.
                                  services.stockquote.StockQuote/"
   xmlns:cexns="http://www.xmethods.net/sd/
                                   CurrencyExchangeService.wsdl"
   xmlns:calns="http://tempuri.org/services/calc">

   <import namespace="http://www.themindelectric.com/wsdl/
        net.xmethods.services.stockquote.StockQuote/" location=
        "http://services.xmethods.net/soap/urn:xmethods-delayed-
                                                    quotes.wsdl"/>

  <import namespace="http://www.xmethods.net/sd/
       CurrencyExchangeService.wsdl" location="http://
       www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl"/>

  <import namespace="http://tempuri.org/services/calc"
          location="http://138.40.91.72:8080/wstk/
                                  Calculator/Calculator.wsdl"/>

  <message name="getCountryNameRequest">
    <part name="country" type="xsd:string"/>
  </message>

  <message name="getCountryNameResponse">
  </message>

  <portType name="QuoteTrackerPortType">
     <operation name="getCountryName">
       <input message="tns:getCountryNameRequest"/>
```

```
        <output message="tns:getCountryNameResponse"/>
      </operation>
  </portType>

<slnk:serviceLinkType name="QuoteFinderLinkType">
    <slnk:role name="QuoteFinder">
      <portType name="qns:net.xmethods.services.
                    stockquote.StockQuotePortType"/>
    </slnk:role>
  </slnk:serviceLinkType>


  <slnk:serviceLinkType name="currencyExchangerLinkType">
    <slnk:role name="currencyExchanger">
      <portType name="cexns:CurrencyExchangePortType"/>
    </slnk:role>
  </slnk:serviceLinkType>

  <slnk:serviceLinkType name="calculatorLinkType">
    <slnk:role name="calculator">
      <portType name="calns:Calculator"/>
    </slnk:role>
  </slnk:serviceLinkType>

  <property name="q_id" type="xsd:string"/>

  <propertyAlias propertyName="tns:q_id"
   messageType="qns:getQuoteRequest1" part="symbol" query="/symbol"/>

  <!-- The service name and the TNS represent my service ID QName -->
  <service name="QuoteTrackerServiceBP"/>

</definitions>
```

E.3 WSDL Specification of the Simple Calculator Service (SCS)

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace=http://tempuri.org/services/calc
      xmlns=http://schemas.xmlsoap.org/wsdl/
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl="http://tempuri.org/services/calc"
      xmlns:intf="http://tempuri.org/services/calc"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <wsdl:message name="addRequest">
      <wsdl:part name="in0" type="xsd:float"/>
      <wsdl:part name="in1" type="xsd:float"/>
   </wsdl:message>

   <wsdl:message name="divResponse">
      <wsdl:part name="divReturn" type="xsd:float"/>
   </wsdl:message>

   <wsdl:message name="multResponse">
      <wsdl:part name="multReturn" type="xsd:float"/>
   </wsdl:message>
```

```
<wsdl:message name="multRequest">
   <wsdl:part name="in0" type="xsd:float"/>
   <wsdl:part name="in1" type="xsd:float"/>
</wsdl:message>

<wsdl:message name="subResponse">
   <wsdl:part name="subReturn" type="xsd:float"/>
</wsdl:message>

<wsdl:message name="subRequest">
   <wsdl:part name="in0" type="xsd:float"/>
   <wsdl:part name="in1" type="xsd:float"/>
</wsdl:message>

<wsdl:message name="addResponse">
   <wsdl:part name="addReturn" type="xsd:float"/>
</wsdl:message>

<wsdl:message name="divRequest">
   <wsdl:part name="in0" type="xsd:float"/>
   <wsdl:part name="in1" type="xsd:float"/>
</wsdl:message>

<wsdl:portType name="Calculator">
   <wsdl:operation name="add" parameterOrder="in0 in1">
      <wsdl:input message="impl:addRequest" name="addRequest"/>
      <wsdl:output message="impl:addResponse" name="addResponse"/>
   </wsdl:operation>

   <wsdl:operation name="sub" parameterOrder="in0 in1">
      <wsdl:input message="impl:subRequest" name="subRequest"/>
      <wsdl:output message="impl:subResponse" name="subResponse"/>
   </wsdl:operation>

   <wsdl:operation name="mult" parameterOrder="in0 in1">
      <wsdl:input message="impl:multRequest" name="multRequest"/>
      <wsdl:output message="impl:multResponse"
                                          name="multResponse"/>
   </wsdl:operation>

   <wsdl:operation name="div" parameterOrder="in0 in1">
      <wsdl:input message="impl:divRequest" name="divRequest"/>
      <wsdl:output message="impl:divResponse" name="divResponse"/>
   </wsdl:operation>

</wsdl:portType>

<wsdl:binding name="CalculatorServiceSoapBinding"
                                          type="impl:Calculator">

   <wsdlsoap:binding style="rpc"
               transport="http://schemas.xmlsoap.org/soap/http"/>

   <wsdl:operation name="add">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="addRequest">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/calc"
                                              use="encoded"/>
      </wsdl:input>
```

```
      <wsdl:output name="addResponse">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://tempuri.org/services/calc"
                                                  use="encoded"/>
      </wsdl:output>
   </wsdl:operation>

   <wsdl:operation name="sub">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="subRequest">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/calc"
                                                  use="encoded"/>
      </wsdl:input>

      <wsdl:output name="subResponse">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         namespace="http://tempuri.org/services/calc"
                                                 use="encoded"/>
      </wsdl:output>
   </wsdl:operation>

   <wsdl:operation name="mult">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="multRequest">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/calc"
          use="encoded"/>
      </wsdl:input>

      <wsdl:output name="multResponse">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/calc"
          use="encoded"/>
      </wsdl:output>
   </wsdl:operation>

   <wsdl:operation name="div">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="divRequest">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/calc"
                                                 use="encoded"/>
      </wsdl:input>

      <wsdl:output name="divResponse">
         <wsdlsoap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://tempuri.org/services/calc"
          use="encoded"/>
      </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
```

```
   <wsdl:service name="CalculatorService">
      <wsdl:port binding="impl:CalculatorServiceSoapBinding"
                                    name="CalculatorService">
         <wsdlsoap:address location="http://138.40.91.72:8080/wstk/
                           services/CalculatorService"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

E.4 WSDL Specification of the Currency Exchanger Service (CES)

```
<?xml version="1.0"?>
<definitions name="CurrencyExchangeService"
     targetNamespace="http://www.xmethods.net/
                                sd/CurrencyExchangeService.wsdl"
     xmlns:tns="http://www.xmethods.net/
                                sd/CurrencyExchangeService.wsdl"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns="http://schemas.xmlsoap.org/wsdl/">

      <message name="getRateRequest">
            <part name="country1" type="xsd:string"/>
            <part name="country2" type="xsd:string"/>
      </message>

      <message name="getRateResponse">
            <part name="Result" type="xsd:float"/>
      </message>

      <portType name="CurrencyExchangePortType">
            <operation name="getRate">
                  <input message="tns:getRateRequest" />
                  <output message="tns:getRateResponse" />
            </operation>
      </portType>

      <binding name="CurrencyExchangeBinding"
                           type="tns:CurrencyExchangePortType">
            <soap:binding style="rpc"
               transport="http://schemas.xmlsoap.org/soap/http"/>
            <operation name="getRate">
                  <soap:operation soapAction=""/>
                  <input >
                        <soap:body use="encoded"
                           namespace="urn:xmethods-CurrencyExchange"
                           encodingStyle="http://schemas.xmlsoap.org/
                                                soap/encoding/"/>
                  </input>

                  <output >
                        <soap:body use="encoded"
                           namespace="urn:xmethods-CurrencyExchange"
                           encodingStyle="http://schemas.xmlsoap.org/
                                                  soap/encoding/"/>
                  </output>
            </operation>
      </binding>

      <service name="CurrencyExchangeService">
```

```
            <port name="CurrencyExchangePort"
                        binding="tns:CurrencyExchangeBinding">
                <soap:address
                    location="http://services.xmethods.net:80/soap"/>
            </port>
        </service>
</definitions>
```

E.5 WSDL Specification of the Stock Quote Service (SQS)

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions name='net.xmethods.services.stockquote.StockQuote'
         targetNamespace='http://www.themindelectric.com/wsdl/
                  net.xmethods.services.stockquote.StockQuote/'
         xmlns:tns='http://www.themindelectric.com/wsdl/
                 net.xmethods.services.stockquote.StockQuote/'
         xmlns:electric='http://www.themindelectric.com/'
         xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
         xmlns:xsd='http://www.w3.org/2001/XMLSchema'
         xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
         xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
         xmlns='http://schemas.xmlsoap.org/wsdl/'>

    <message name='getQuoteResponse1'>
       <part name='Result' type='xsd:float'/>
    </message>

    <message name='getQuoteRequest1'>
       <part name='symbol' type='xsd:string'/>
    </message>

    <portType name='net.xmethods.services.
                          stockquote.StockQuotePortType'>
        <operation name='getQuote' parameterOrder='symbol'>
          <input message='tns:getQuoteRequest1'/>
          <output message='tns:getQuoteResponse1'/>
        </operation>
    </portType>

    <binding name='net.xmethods.services.stockquote.
       StockQuoteBinding' type='tns:net.xmethods.services.
                                stockquote.StockQuotePortType'>
         <soap:binding style='rpc'
                transport='http://schemas.xmlsoap.org/soap/http'/>
            <operation name='getQuote'>
              <soap:operation soapAction='urn:xmethods-delayed-
                                          quotes#getQuote'/>
                <input>
                   <soap:body use='encoded' namespace='urn:xmethods-
                        delayed-quotes' encodingStyle=
                        'http://schemas.xmlsoap.org/soap/encoding/'/>
                </input>

                <output>
                   <soap:body use='encoded' namespace='urn:xmethods-
                        delayed-quotes' encodingStyle=
                        'http://schemas.xmlsoap.org/soap/encoding/'/>
                </output>
            </operation>
     </binding>
```

```
    <service name='net.xmethods.services.stockquote.
                                    StockQuoteService'>

      <port name='net.xmethods.services.stockquote.StockQuotePort'
            binding='tns:net.xmethods.services.
                                     stockquote.StockQuoteBinding'>
           <soap:address location='http://64.124.140.30:9090/soap'/>
      </port>
    </service>
</definitions>
```

Appendix F

Specification of the RateTrackerProcess

F.1 BPEL Specification of the RateTrackerProcess

```
<process name="rateTrackerProcess"
      targetNamespace="http://tempuri.org/services/PriceTrackerBpel"
      suppressJoinFailure="yes"
      xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:wsdlns="http://tempuri.org/services/PriceTrackerWsdl"
      xmlns:cexns="http://www.atlaz.net/webservices">

   <partners>
      <partner name="customer"
               serviceLinkType="wsdlns:rateTrackerLinkType"
               myRole="rateTracker"/>
      <partner name="currencyExchanger"
               serviceLinkType="wsdlns:currencyExchangerLinkType"
               partnerRole="currencyExchanger"/>
   </partners>

   <variables>
      <variable name="conversionRequest"
                messageType="wsdlns:getConversionRequest"/>
      <variable name="conversionResponse"
                messageType="wsdlns:getConversionResponse"/>
      <variable name="amountRequest"
                messageType="wsdlns:getAmountRequest"/>
      <variable name="amountResponse"
                messageType="wsdlns:getAmountResponse"/>
      <variable name="rateRequest"
                messageType="cexns:getRateRequestSoapIn"/>
      <variable name="rateResponse"
                messageType="cexns:getRateRequestSoapOut"/>
    </variables>

    <sequence>
        <receive name="getAmount" partner="customer"
           portType="wsdlns:RateTrackerPortType"
           operation="getAmount" variable="amountRequest"
                                      createInstance="yes"/>
        <switch name="checkAmount">
            <case name="amountNegative"
                  condition="bpws:getVariableData('amountRequest',
                                'amount') &lt; 0">
               <sequence>
                  <assign name="assign1">
                    <copy>
                        <from expression="0"/>
                        <to variable="amountResponse" part="amount"/>
                    </copy>
                  </assign>
                  <reply name="negative-reply" partner="customer"
                      portType="wsdlns:RateTrackerPortType"
                      operation="getAmount"
                      variable="amountResponse"/>
                    </sequence>
```

```
</case>
<case name="amountPositive"
         condition="bpws:getVariableData('amountRequest',
                                         'amount') &gt; 0">
      <pick name="getConversion">
        <onMessage name="conversion" partner="customer"
            portType="wsdlns:RateTrackerPortType"
            operation="getConversion"
            variable="conversionRequest">
            <sequence>
              <assign name="assign2">
                <copy>
                   <from variable="conversionRequest"
                                    part="country1"/>
                   <to variable="rateRequest"
                                    part="currency1"/>
                </copy>
                <copy>
                   <from variable="conversionRequest"
                                     part="country2"/>
                   <to variable="rateRequest"
                                   part="currency2"/>
                </copy>
                <copy>
                   <from variable="amountRequest"
                               part="amount"/>
                   <to variable="rateRequest"
                                  part="amount"/>
                </copy>
               </assign>
               <invoke name="requestRate"
                 partner="currencyExchanger"
                 portType="cexns:GetCurrencyExchangeSOAP"
                 operation="getRateRequest"
                 inputVariable="rateRequest"
                 outputVariable="rateResponse"/>

                <assign name="assign3">
                   <copy>
                        <from variable="rateResponse"
                                    part="value"/>
                        <to variable="conversionResponse"
                                     part="amount"/>
                     </copy>
                 </assign>

                 <reply name="reply-Conversion"
                   partner="customer"
                   portType="wsdlns:RateTrackerPortType"
                   operation="getConversion"
                   variable="conversionResponse"/>
            </sequence>
          </onMessage>
          <onAlarm name="noConversion" for="'PT30S'">
             <sequence>
               <assign name="assign4">
                 <copy>
                     <from variable="amountRequest"
                                     part="amount"/>
                     <to variable="amountResponse"
                                     part="amount"/>
```

```
                        </copy>
                      </assign>
                      <reply name="reply-amount"
                            partner="customer"
                            portType="wsdlns:RateTrackerPortType"
                            operation="getAmount"
                            variable="amountResponse"/>
                      </sequence>
                    </onAlarm>
                  </pick>
              </case>
        </switch>
    </sequence>
</process>
```

F.2 WSDL Specification of the RateTrackerProcess

```
<?xml version="1.0"?>
<definitions
     targetNamespace="http://tempuri.org/services/rateTrackerWsdl"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:slnk="http://schemas.xmlsoap.org/ws/2003/03/service-link/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:tns="http://tempuri.org/services/PriceTrackerWsdl"
     xmlns:cexns="http://www.atlaz.net/webservices">

  <import namespace="http://www.atlaz.net/webservices"
       location=
       "http://www.atlaz.net/webservices/GetCurrencyExchange.wsdl"/>

  <message name="getConversionRequest">
    <part name="country1" type="xsd:string"/>
    <part name="country2" type="xsd:string"/>
  </message>

  <message name="getConversionResponse">
    <part name="result" type="xsd:float"/>
  </message>

  <message name="getAmountRequest">
    <part name="amount" type="xsd:float"/>
  </message>

  <message name="getAmountResponse">
    <part name="amount" type="xsd:float"/>
  </message>

  <portType name="RateTrackerPortType">
     <operation name="getConversion">
       <input message="tns:getConversionRequest"/>
       <output message="tns:getConversionResponse"/>
     </operation>

     <operation name="getAmount">
       <input message="tns:getAmountRequest"/>
       <output message="tns:getAmountResponse"/>
     </operation>
  </portType>
```

```
  <slnk:serviceLinkType name="rateTrackerLinkType">
    <slnk:role name="rateTracker">
      <portType name="tns:RateTrackerPortType"/>
    </slnk:role>
  </slnk:serviceLinkType>

  <slnk:serviceLinkType name="currencyExchangerLinkType">
    <slnk:role name="currencyExchanger">
      <portType name="cexns:GetCurrencyExchangeSOAP"/>
    </slnk:role>
  </slnk:serviceLinkType>

  <slnk:serviceLinkType name="calculatorLinkType">
    <slnk:role name="calculator">
      <portType name="calns:Calculator"/>
    </slnk:role>
  </slnk:serviceLinkType>

  <!-- The service name and the TNS represent my service ID QName -->
  <service name="rateTrackerServiceBP"/>


</definitions>
```

F.3 WSDL Specification of GetCurrencyExchange

```
<?xml version="1.0"?>
<definitions
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:si="http://soapinterop.org/xsd"
    xmlns:tns="http://mywebservices.fr.st/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://www.atlaz.net/webservices">

    <message name="getRateRequestSoapOut">
        <part name="value" type="xsd:float" />
    </message>

    <message name="getRateRequestSoapIn">
        <part name="number" type="xsd:float" />
        <part name="currency1" type="xsd:string" />
        <part name="currency2" type="xsd:string" />
    </message>

    <portType name="GetCurrencyExchangeSOAP">
        <operation name="getRateRequest">
            <input message="getRateRequestSoapIn" />
            <output message="getRateRequestSoapOut" />
        </operation>
    </portType>


    <binding name="GetCurrencyExchange" type="GetCurrencyExchange">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
        <operation name="getRateRequest">
         <soap:operation
        soapAction="http://www.atlaz.net/webservices/webservices.php"
                                            style="document"/>
            <input name="getRateRequestInput">
                <soap:body use="literal"/>
            </input>
            <output name="getRateRequestOutput">
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="GetCurrencyExchangeSOAP">
        <port binding="GetCurrencyExchangeSOAPSOAP"
                     name="GetCurrencyExchangeSOAPSOAPPort">
          <soap:address
              location="http://www.atlaz.net/webservices/
                                GetCurrencyExchange.php"/>
        </port>
    </service>
</definitions>
```

Appendix G

Guidelines for Using the Prototype of the Monitoring Framework

G.1 The Monitoring Tool

The Monitoring tool allows a service provider to monitor the behavioural properties, functional properties and quality constraints of a service based system. The monitoring tool is based on the architecture discussed in Chapter 5. Binary distribution of The monitoring tool can be downloaded from

http://www.soi.city.ac.uk/~am697/monitoring_tool/City_Monitor.html

This binary distribution composed of two main components,

- The Analyzer
- The Manager

These two components wrap up all the components of the architecture discussed in Chapter 5. The analyzer wraps up the *monitor* along with the *event database handler* and the *formula database handler*. The manager wraps up the *monitoring console*, the *event receiver* and the *monitor manager*. In this distribution the analyzer provides the basic monitoring functionality and it is deployed as web service. The manager is a client of the analyzer and it provides GUI based front end to the end user.

In this appendix we discuss the installation of the The monitoring tool and provide instructions to use the tool. This user manual assumes that Windows is the operating system of the target machine.

G.2 Required Software

Following tools are needed to use the monitoring tool

- Tomcat server - which can be downloaded from http://tomcat.apache.org/. Installation guide for tomcat server is also available there. We tested our latest implementation with tomcat version 5.0.14

- Axis server – This server can be downloaded from http://ws.apache.org/axis/. An installation guide for the server is also available at the same site. We tested our implementation with Axis version 1.4.
- Bpws4j process execution engine - which can be downloaded from http://alphaworks.ibm.com/tech/bpws4j. Installation guide for this tool is also available there. We tested our implementation with bpws4j version 2.

G.3 Installation of Required Software

In this section we briefly describe how to install the required software and how to use them. Detail user manual of the software can be found in the developer sites of the respective software.

Tomcat Server

1 Download a Java Development Kit (JDK) from: http://java.sun.com/j2se/. Install the JDK according to the instructions included with the release. Set an environment variable JAVA_HOME to the pathname of the directory into which JDK release has been installed.

2 Download the tomcat binary distribution from http://tomcat.apache.org/. Unpack the binary distribution into a convenient location so that the distribution resides in its own directory. In the rest of the manual we assume that tomcat has been installed at C:\tomcat. Please consult the release note of tomcat for the selection of right XML. You may need to copy xalan.jar file in the C:\tomcat\common\endorsed folder and activation.jar file in the C:\tomcat\common\lib folder. These files are included in the "jars" folder of the binary distribution of the the monitoring tool (see below). These archives can also be downloaded from www.appache.org.

3 Tomcat can be started by executing the following command, c:\tomcat\bin\startup.bat. After start up, the default web applications included with Tomcat will be available by visiting: http://localhost:8080/

4 Tomcat can be shut down by executing the following command: c:\tomcat\bin\shutdown

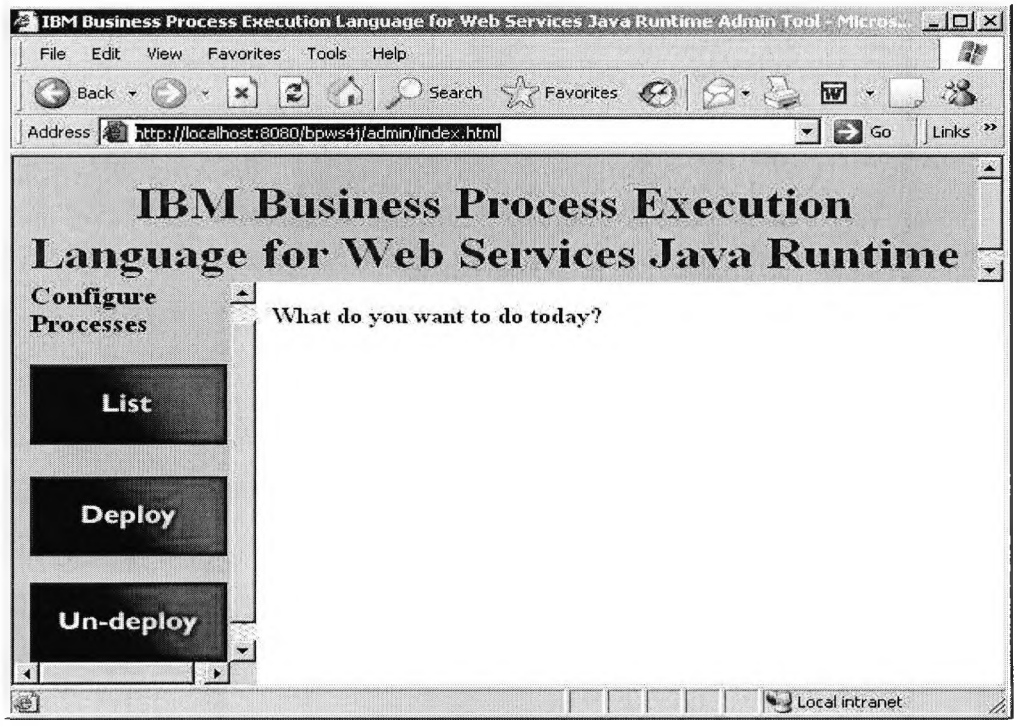
Axis Server

1. Download the axis binary distribution from http://ws.apache.org/axis/ and extract it at location of your choice. In the rest of the manual we assume that axis has been extracted at C:\axis_extracted. Copy the folder named C:\axis_extracted\webapps\axis into C:\tomcat\webapps. Start tomcat by executing C:\tomcat\bin\startup.bat and using a web

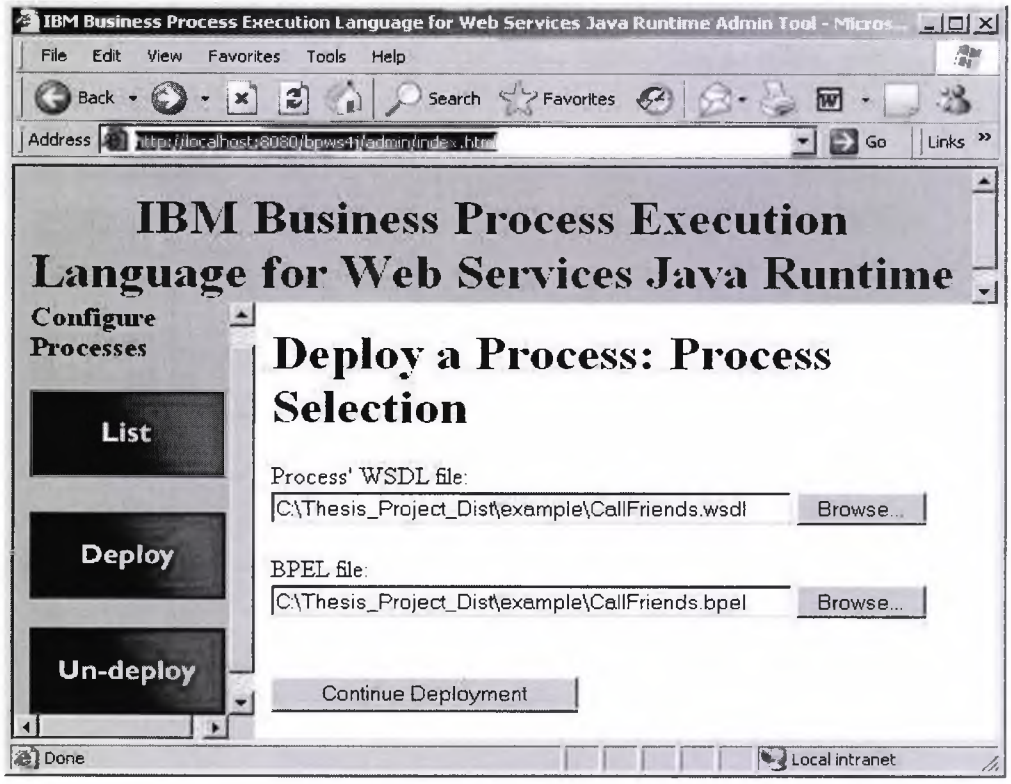
browser visit http://localhost:8080/axis/happyaxis.jsp. If happyaxis.jsp shows message that it cannot find some packages, then copy the relevant packages from C:\axis_extracted\webapps\axis\WEB-INF\lib to C:\tomcat\common\lib and restart tomcat.

Bpws4j Process Execution Engine

1. Download the binary distribution of the bpws4j engine from http://alphaworks.ibm.com/tech/bpws4j and extract it at location of your choice. In this manual we assume that bpws4j has been extracted at C:\bpws4j_extracted. You have to copy mail.jar file in C:\tomcat\common\lib. This file is included in the "jars" folder of the binary distribution of the the monitoring tool (see below).
2. Copy the file bpws.war from C:\bpws4j_extracted\webapps to C:\tomcat\webapps. Start tomcat by executing C:\tomcat\bin\startup.bat and using a web browser visit http://localhost:port/bpws4j/soaprpcrouter. The browser should display "Sorry, I don't speak via HTTP GET- you have to use HTTP POST to talk to me". If you don't see this message, your server is not configured correctly and consult the user manual comes with bpws4j distribution.
3. To manage BPEL processes, using a web browser visit the page http://localhost:8080/bpws4j/admin/index.html. The page looks like as shown below,

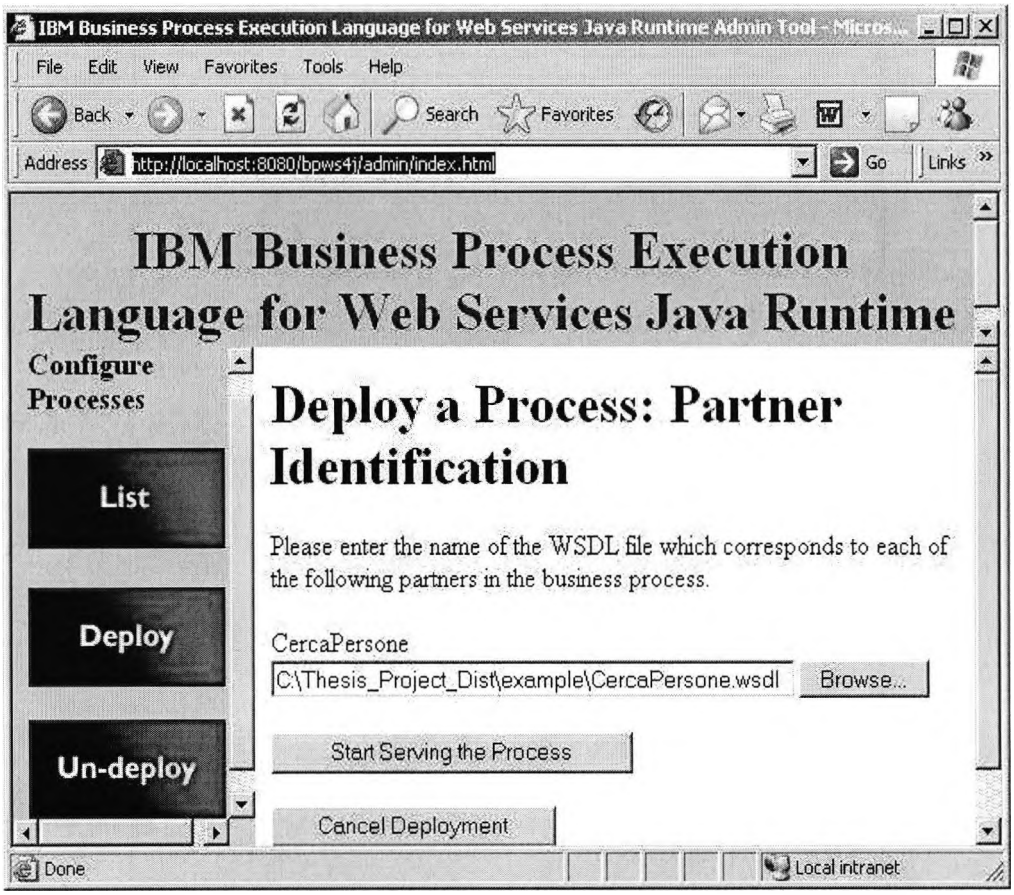


If you click on the "List" button, you will be presented with a list of all of the processes which are currently deployed to the engine. To deploy a new process, click on the "Deploy" button. A new page will appear, which looks as shown below,



Select the WSDL and BPEL file of the process to be deployed, then click on the "Continue Deployment" button. A new page will appear as shown below. In the new page select WSDL files (only one in this example) for all the services deployed by the BPEL process. Click on the "Start Serving the Process" button. A new page will appear to confirm the successful deployment of the process.

To un-deploy a deployed process, click on the "Un-deploy" button. A list of deployed processes will appear. Click on the process name that should be un-deployed.



4 To configure the bpws4j engine to send log events to a specific port (i.e. the port number that the event listener is listening to), open the C:\tomcat\webapps\bpws4j\WEB-INF\classes\ log4j.properties file in a text editor (e.g. norepad.exe). Make sure the following lines are added in the log4j.properties file,

```
log4j.rootLogger=DEBUG, A1
log4j.appender.A1=org.apache.log4j.net.SocketAppender
log4j.appender.A1.Port= DESTINATION_PORT_NUMBER
log4j.appender.A1.RemoteHost= DESTINATION_IP_ADDRESS
```

Where `DESTINATION_IP_ADDRESS` is the IP address of the host on which event receiver is running and `DESTINATION_PORT_NUMBER` is the port number to which the *event receiver* is listening. Now restart tomcat.

G.4 Installation of the Monitoring Tool

Download the binary distribution of the monitoring tool from http://www.soi.city.ac.uk/~am697/monitoring_tool/City_Monitor.html and unzip the City_Monitor.zip at location of your choice. In this manual we assume that bpws4j has been extracted at C:\City_Monitor. There are five folders inside the C:\City_Monitor folder. These are,

- Manager – This folder contains the manager.
- Jars – This folder contains all the jar files needed to run the monitoring tool.
- Example – This folder contains an example BPEL process and example policy file for the BPEL process.
- BPEL_Client – This folder contains a client for the example BPEL process.
- Analyzer – This folder contains the analyzer that provides the monitoring service. To install the analyzer, copy the folder C:\City_Monitor\analyzer\code in the C:\tomcat\webapps\axis\WEB-INF\classes folder. Start tomcat by executing the command C:\tomcat\bin. In a command prompt window execute the command C:\City_Monitor\analyzer\deploy.bat. The analyzer service is up and the wsdl specification of the analyzer service can be seen at: http://localhost:8080/axis/services/analyzerService?wsdl, and the analyzer service endpoint is http://localhost:8080/axis/services/analyzerService

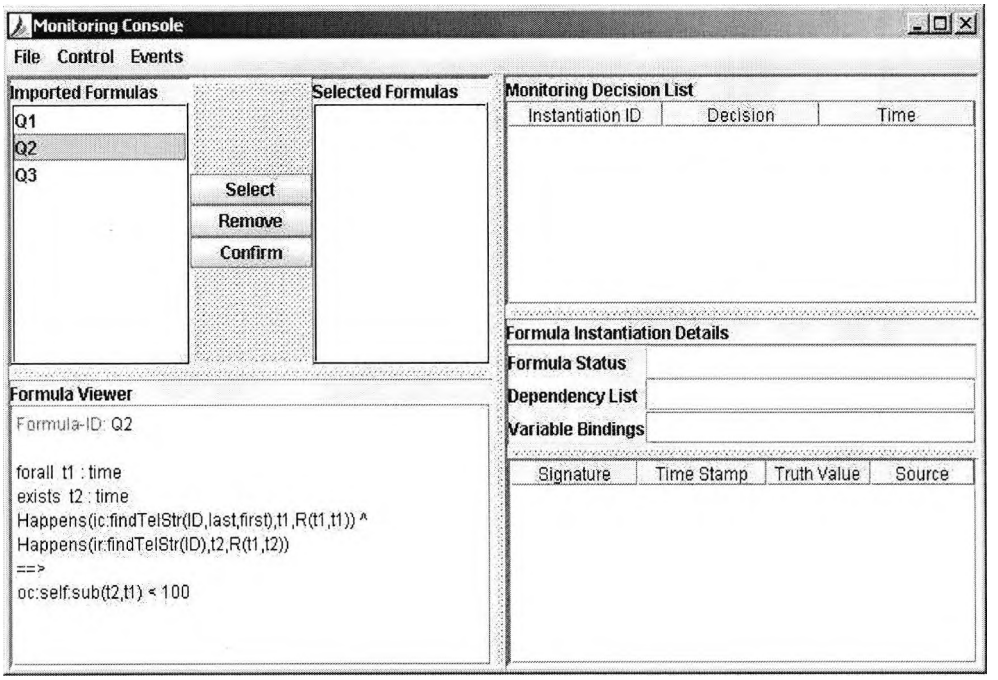
G.5 Accessing the Analyzer Service Using the Manager

The monitoring manager is used to import monitoring policy and select the formulas to be monitored, send the selected formulas to the analyzer, start the event receiver for a monitoring session, initiate a polling process that retrieves possible violations of the properties and view the result of monitoring. To retrieve violations of properties, the monitoring manager polls the analyzer at regular time intervals that has been specified by the user in the policy and shows the results that it retrieves in a formula viewer.

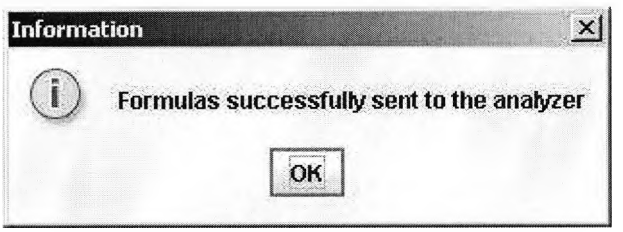
To use the monitor manager, follow the following steps

(i) To start the monitor manager, in a command prompt window execute the command C:\City_Monitor\manager\RunManager. Following this, the monitor manager window will pop up.

(ii) Then, to import the monitoring policy, select the option "Import Policy" from the "File" menu of the manager. In the file opening dialog box that appears, choose an XML file that contains the monitoring policy. The monitor manager will then read the monitoring policy from the file and display the formulas to be monitored as shown in the figure below.



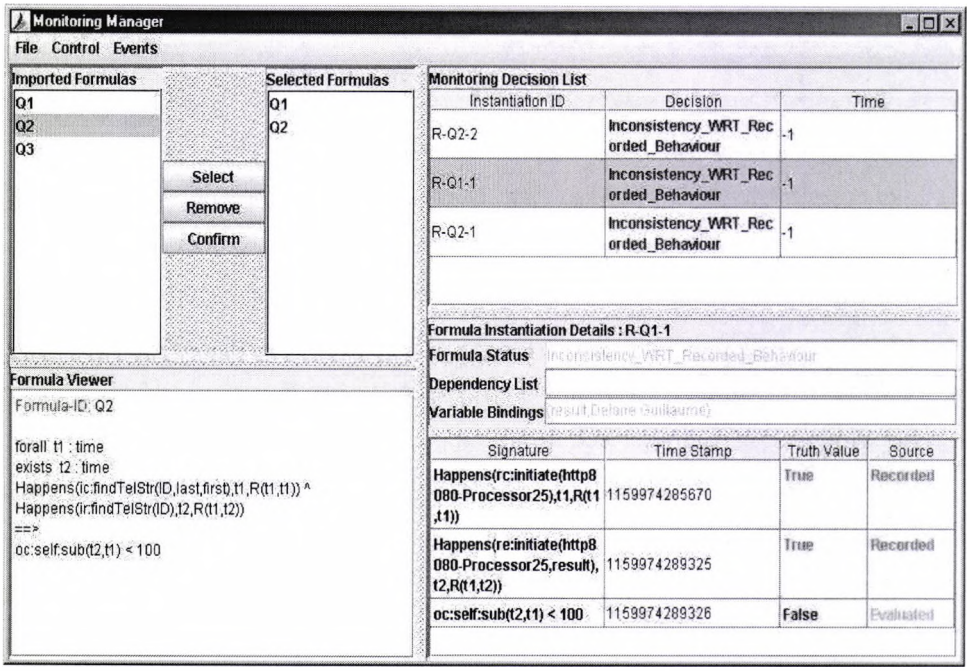
The monitoring manager lists the identifiers of the imported formulas in the "Imported Formulas" panel. To view a formula in the event calculus format, the user may select its ID. Following this selection, the formula with the selected ID will be shown in the "Formula Viewer" panel of the manager. If the user wants to select the formula to be monitored, he/she may select its ID in the imported formulas panel and click on the "Select" button. Following this, the selected formula will appear in the "Selected Formulas" panel. The user may repeat the same process to select more formulas. When the selection is complete, the user can click on the "Confirmed" button, to send the formulas to the data analyser. If the submission of formulas to the analyser is successful, the monitor manager will show the following message. The user should press the "Ok" button to continue.



(iii) The next step is to provide the analyzer with runtime events. The user should select the option "Start Event Receiver" from "Control" menu. At this point the user has to

start the BPEL process to be monitored. The user may stop the event receiver by selecting the option "Stop Event Receiver" in the "Control" menu[1].

(iv) To start polling the analyzer in order to view the violations of the formulas being monitored, the user should select the option "start polling" from the control menu. Following this, the manager will start polling the data analyzer at regular time intervals specified in the monitoring policy. The manager shows the list of instances of the violated and satisfied formulas in the "Monitoring Decision List" panel as shown in the figure below. This panel will be updated at the regular intervals. The Monitoring Decision List will show the monitoring summary of each instance of each formula. The left most column in this list shows the unique formula instance ID, the middle column shows the decision for the formula instance, and the right most column shows the time when the decision was made by the analyzer.



To view the details of a formula instance, the user should select the relevant formula instance in the Monitoring Decision List. Following this, the manager shows the details of the formula instance in the "Formula Instantiation Details" panel. This panel displays the formula status, other formulas that the specific formula may depend on, and the values bound to the variables of the formula. "Formula Instantiation Details" panel also shows the truth values of the individual predicates of

[1] Once the event receiver is stopped the socket established with the tomcat server is lost. User needs to restart the whole process (restart tomcat, and monitor manager) if s/he wants to restart the event receiver.

the formula, the timestamps of the establishment of these truth values, and the source of the information that underpins them.

G.5.1 Example

This distribution includes a simple BPEL process that can be used to check the functionality of the analyzer and the manager. To make use of this example follow the following steps,

- Install bpws4j engine and configure it to send log events to port 12345 as described in Section G.3.
- Start tomcat server.
- Deploy the example BPEL process in the bpws4j engine as described in Section G.3, using the files c:\City_Monitor\example\CallFriends.bpel, c:\City_Monitor\example\CallFriends.wsdl, c:\City_Monitor\example\CercaPersone.wsdl
- Open the policy file policy.xml in the c:\City_Monitor\example folder using the manager. Select the formulas to be monitored and send the formulas to the data analyzer by clicking the confirmed button as described in G.5.
- Start the event receiver from the manager.
- In a command prompt window give the command C:\City_Monitor\BPEL_Client_IBM\RunIBMBPELClientt. This starts a client of the BPEL process that contacts the BPEL process for 10 minutes, with a random interval of up to 10 seconds between each interaction.
- Start polling from the manager, it opens the decision viewer window and updates it regularly.

G.6 Accessing the Analyzer Service as a Web Service

As described in Section G.1 and G.4 the analyzer is deployed as a web service, this service can be accessed without using the manager comes with this distribution. In this section we discuss the interface of the analyzer web service. The WSDL file of the analyzer service is available at http://localhost:8080/axis/services/analyzerService?wsdl, and the analyzer service endpoint is http://localhost:8080/axis/services/analyzerService. Following figure shows the WSDL file of the analyzer service,

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://localhost:8080/axis/services/analyzerService"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://localhost:8080/axis/services/analyzerService"
xmlns:intf="http://localhost:8080/axis/services/analyzerService" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->

  <wsdl:message name="checkRuleRequest"> <wsdl:part name="monitoringRuleId" type="soapenc:string"/> </wsdl:message>
  <wsdl:message name="monitorRuleRequest"> <wsdl:part name="input" type="soapenc:string"/> </wsdl:message>
  <wsdl:message name="notifyResponse"> <wsdl:part name="notifyReturn" type="soapenc:string"/> </wsdl:message>
  <wsdl:message name="checkRuleResponse"> <wsdl:part name="checkRuleReturn" type="soapenc:string"/> </wsdl:message>
  <wsdl:message name="notifyRequest"> <wsdl:part name="monitoringDatum" type="soapenc:string"/> </wsdl:message>
  <wsdl:message name="monitorRuleResponse"> <wsdl:part name="monitorRuleReturn" type="xsd:boolean"/>
  </wsdl:message>

  <wsdl:portType name="DataAnalyzer">
    <wsdl:operation name="notify" parameterOrder="monitoringDatum">
      <wsdl:input message="impl:notifyRequest" name="notifyRequest"/>
      <wsdl:output message="impl:notifyResponse" name="notifyResponse"/>
    </wsdl:operation>
    <wsdl:operation name="monitorRule" parameterOrder="input">
      <wsdl:input message="impl:monitorRuleRequest" name="monitorRuleRequest"/>
      <wsdl:output message="impl:monitorRuleResponse" name="monitorRuleResponse"/>
    </wsdl:operation>
    <wsdl:operation name="checkRule" parameterOrder="monitoringRuleId">
      <wsdl:input message="impl:checkRuleRequest" name="checkRuleRequest"/>
      <wsdl:output message="impl:checkRuleResponse" name="checkRuleResponse"/>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="analyzerServiceSoapBinding" type="impl:DataAnalyzer">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="notify">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="notifyRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://code" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="notifyResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://localhost:8080/axis/services/analyzerService" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="monitorRule">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="monitorRuleRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://code" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="monitorRuleResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                    namespace="http://localhost:8080/axis/services/analyzerService" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="checkRule">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="checkRuleRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://code" use="encoded"/>
      </wsdl:input>
      <wsdl:output name="checkRuleResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                   namespace="http://localhost:8080/axis/services/analyzerService" use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="DataAnalyzerService">
    <wsdl:port binding="impl:analyzerServiceSoapBinding" name="analyzerService">
      <wsdlsoap:address location="http://localhost:8080/axis/services/analyzerService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

According to the WSDL file the interface of the analyzer service exposes three methods. In the following we discuss these methods with the data types,

- *monitorRule* – This method is used to send the formulas to be monitored to the analyzer. The input parameter to this method is of type string, more specifically the string representation of the XML formulas to be monitored. The formulas to be monitored are expressed in XML according to the schema presented in appendix A. The return type of this method is Boolean.
- *notify* – This method is used to send a monitoring event to the analyzer. This method has one input parameter of type string, which is the string representation of an event expressed in XML. The event is expressed in XML according to the following schema,

```
<?xml version="1.0" encoding="UTF-8" ?>
 <xs:schema targetNamespace="http://tempuri.org/secse/event"
       xmlns="http://tempuri.org/secse/event"
       xmlns:xs="http://www.w3.org/2001/XMLSchema"  elementFormDefault="qualified">

  <!-- define event -->

  <xs:element name="event" type="eventType"/>
  <!-- definition of complex and simple types -->
  <xs:complexType name="eventType">
     <xs:sequence>
       <xs:element name="timeStamp" type="xs:string"/>
       <xs:element name="id" type="xs:string"/>
       <xs:element name="prefix" type="xs:string"/>
       <xs:element name="partnerID" type="xs:string"/>
       <xs:element name="operationName" type="xs:string"/>
       <xs:element name="variable" type="variableType" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
  </xs:complexType>

  <xs:complexType name="variableType">
     <xs:sequence>
       <xs:element name="name" type="xs:string"/>
       <xs:element name="type" type="xs:string"/>
       <xs:element name="value" type="xs:string"/>
     </xs:sequence>
  </xs:complexType>

</xs:schema>
```

- *checkRule* – This method is used to poll the analyzer for monitoring result of a specific formula. This method has one input parameter of type string which signifies the ID of a formula. The return type of this method is string, which is the string representation of all the instances of a formula. Formula instances are expressed in XML according to the schema described in Appendix C.