10

Monitoring *WS-Agreement*s: An Event Calculus–Based Approach

Khaled Mahbub and George Spanoudakis

Department of Computing, City University, London, EC1V 0HB
gespan|am697@soi.city.ac.uk

Abstract. In this chapter, we present a framework that we have developed to support the monitoring of service level agreements (SLAs). The agreements that can be monitored by this framework are expressed in an extension of *WS-Agreement* that we propose. The main characteristic of the proposed extension is that it uses an event calculus–based language, called *EC-Assertion*, for the specification of the service guarantee terms in a service level agreement that need to be monitored at runtime. The use of *EC-Assertion* for specifying service guarantee terms provides a well-defined semantics to the specification of such terms and a formal reasoning framework for assessing their satisfiability. The chapter describes also an implementation of the framework and the results of a set of experiments that we have conducted to evaluate it.

10.1 Introduction

The ability to set up and monitor service level agreements (SLAs) has been increasingly recognized as one of the essential preconditions for the deployment of web services [28]. Service level agreements are set through collaboration between service consumers and service producers in order to specify the terms under which a service that is offered to the former by the latter is to be deployed and the quality properties that it should satisfy under these terms. The ability to monitor the compliance of the provision of a service against a service level agreement at runtime is crucial from the point of view of both the service consumer and the service producer.

In the case of service consumers, monitoring service level agreements is necessary due to the need to check if the terms of an agreement are satisfied in a specific operational setting (i.e., the set of the running instances of the services involved in the agreement and the computational resources that these services are deployed on or they use to communicate), identify the consequences that the violation of certain terms in an agreement might have onto their systems, and request the application of any penalties that an agreement prescribes for the violated service provision terms.

For service providers, the monitoring of the provision of a service against the terms specified in an agreement is necessary in order not only to gather evidence regarding the provision, which may be necessary if a dispute with a service consumer arises over the provision, but also to identify problems with the delivery of the service and take action before an agreement is violated. For instance, if an agreement requires that on average a service should respond within N time units over a specific time period, monitoring the performance of the service may spot a performance deviation early enough to give the provider an opportunity to address the problem (by adding, for instance, an extra server at runtime or reducing the level of provision of the same service to other consumers who do not have strictprovision terms).

In this chapter, we describe a framework that we have developed to support the monitoring of functional and quality of service requirements which are specified as part of service level agreements. This framework can monitor the provision of services to service-based software systems (referred to as "SBS" systems, see Fig. 10.1). A for our framework is a system that deploys one or more external web services which are coordinated by a composition

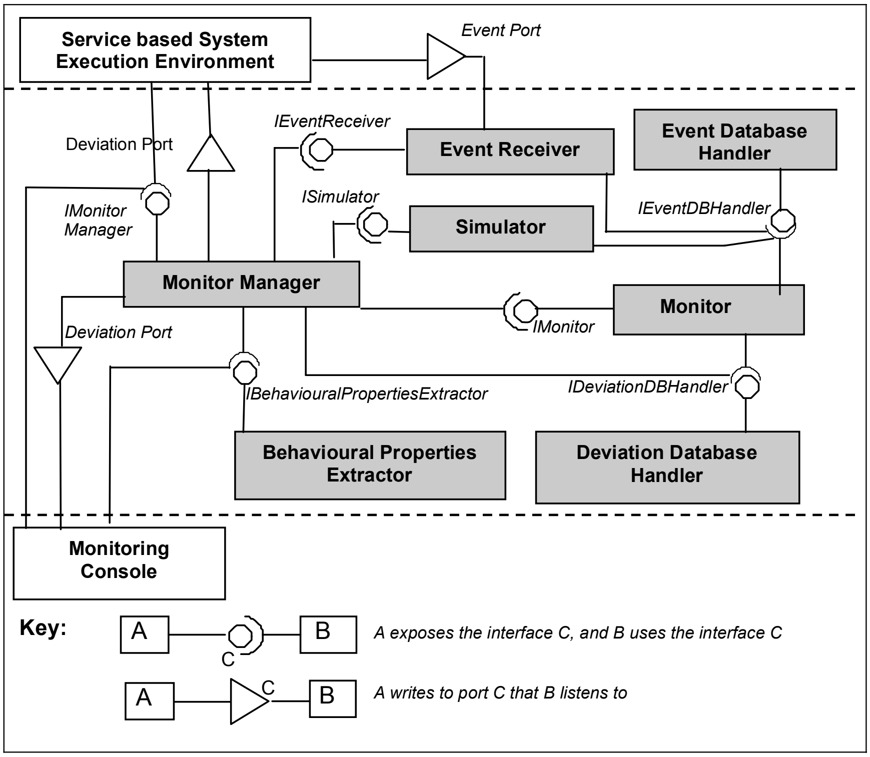

Service based System Execution Environment
Event Port
Deviation Port
IEventReceiver
Event Receiver
Event Database Handler
IMonitor Manager
ISimulator
Simulator
IEventDBHandler
Monitor Manager
Monitor
IMonitor
Deviation Port
IBehaviouralPropertiesExtractor
IDeviationDBHandler
Behavioural Properties Extractor
Deviation Database Handler
Monitoring Console
Key:
A
C
B
A exposes the interface C, and B uses the interface C
A
C
B
A writes to port C that B listens to

Fig. 10.1. Monitoring framework

process that is expressed in BPEL [1]. This composition process provides the required system functionality by calling operations in the external web services, receiving and processing the results that these services return, and accepting and/or responding to requests from them. It should be noted that the external services which are deployed by an SBS system may be interacting directly with third-party services without the intervention of this system. Such interactions are not taken into account during monitoring. Thus, all the external services of an SBS system are effectively treated as atomic services.

The service level agreements that can be monitored by the framework that we present in this chapter are expressed using an extension of *WS-Agreement* [2] that we have defined for this purpose. This extension supports the description of (a) the operational context of an agreement, (b) the policy for monitoring an agreement, and (c) the functional and quality requirements for the service which is regulated by the agreement and need to be monitored (i.e., the guarantee terms in the terminology of *WS-Agreement*). The extensions of *WS-Agreement* that we have introduced to support (a) and (b) have been directly integrated into the XML schema that defines this language. To support the specification of (c), we have developed a new language in which service guarantee terms are specified in terms of (i) events which signify the invocation of operations of a service by the composition process of an SBS system and returns from these executions, (ii) events which signify calls of operations of the composition process of an SBS system by external services and returns from these executions, and (iii) the effects that events of either of the above kinds have on the state of an SBS system or the services that it deploys (e.g., change of the values of system variables). This language has been defined by a separate XML schema and is called . *EC-Assertion*. It is based on (EC) [34] which is a first-order temporal logic language. Specifications of service guarantee terms in *EC-Assertion* can be developed independently of *WS-Agreement* and subsequently referred to by it.

The events which are used in the specification of the service description and guarantee terms in an agreement are restricted to those which can be observed during the execution of the composition process of an SBS system. This set of events is determined by a static analysis of the BPEL composition process of this system that is performed by our framework.

The choice of event calculus (EC) as the language for specifying the service guarantee terms in an agreement 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 monitoring framework has been designed with the objective to support service level agreements. The term “non-intrusive monitoring” in this context 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. Given this definition, non-intrusive monitoring 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 [10] or SBS monitoring by code weaved into BPEL processes [5]). It 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., [19, 33]).

The framework that we present in this chapter is non-intrusive as it is based on events which are captured during the operation of an SBS 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 parallel with it. Furthermore, our framework can monitor different types of deviations from service guarantee terms including: (a) violations of terms by the recorded behavior of a system and (b) violations of service guarantee terms by the expected behavior of the system. These types of deviations were originally defined in [36] and are discussed in this chapter. Additional forms of violations that can be detected by the framework are described in [36].

The framework that we discuss in this chapter was originally developed to support the monitoring of functional service requirements outside the context of *WS-Agreement* and the main formal characteristics of the original form of the framework have been presented in [36]. Hence, in this chapter, our focus is to discuss how this framework can be used to support the monitoring of *WS-Agreement* and introduce the extensions to this standard that enable the use of our framework for this purpose. Furthermore, in this chapter, we present an extension of the specification language of the framework that is based on the use of internal and external operations in event calculus formulas which enable the specification and monitoring of wider range of quality of service requirements.

The rest of the chapter is structured as follows. In Sect. 10.2, we briefly introduce our monitoring framework. In Sect. 10.3, we describe the extensions that we have introduced to *WS-Agreement* in order to specify the service guarantee terms that can be monitored at runtime and policies for performing this monitoring. In Sect. 10.4 we describe the monitoring process that is realized by the framework. In Sect. 10.5, we discuss the prototype that we have developed to implement the framework. In Sect. 10.6, we present the results of an experimental evaluation of the framework. In Sect. 10.7, we review related work. Finally, in Sect. 10.8, we conclude with an overview of our approach and directions for future work.

10.2 Overview of Monitoring Framework

Our framework assumes that the deployment platform of a service-based system is an environment that executes the composition process of the system and can provide the events that will be used during monitoring (see the component *Service Based System Execution Environment* in Fig. 10.1). The framework itself consists of a *monitoring manager*, an *event receiver*, a *monitor*, an *event database*, a *deviation database*, and a *monitoring console*.

The *monitoring manager* is the component that has responsibility for initiating, coordinating, and reporting the results of the monitoring process. Once it receives a request for starting a monitoring activity as specified by the monitoring policy of an agreement, it checks whether it is possible to monitor the service guarantee terms of the agreement as specified in this policy (i.e., given the BPEL process of the SBS system that is identified in the policy and the event reporting capabilities indicated by the type of the execution environment of the SBS system). If the service guarantee terms can be monitored, it starts the event receiver to capture events from the SBS execution environment and passes to it the events that should be collected. It also sends to the monitor the formulas to be checked.

The *event receiver* polls the event port of the SBS execution environment at regular time intervals as specified in the monitoring policy in order to get the stream of events sent to this port. After receiving an event, the event receiver identifies its type and, if it is relevant to the service guarantee terms of the agreement being monitored, it records the event in the event database of the framework. All the events which are not relevant to monitoring are ignored.

The *monitor* retrieves the events which are recorded in the database during the operation of the SBS system in the order of their occurrence, derives (subject to the monitoring mode of an agreement) other possible events that may have happened without being recorded (based on assumptions set for an SBS system in an agreement and its behavioral properties), and checks if the recorded and derived events are compliant with the requirements being monitored. In cases where the recorded and derived events are not consistent with service guarantee terms in an agreement, the monitor records the deviation in a *deviation database*.

The *monitoring manager* polls the deviation database of the framework at regular time intervals to check if there have been any deviations detected with respect to a given monitoring policy and reports them to the port specified by the monitoring policy.

The *behavioral properties extractor* takes as input the BPEL process of the SBS system to be monitored and generates a specification of the behavioral properties of this system in event calculus. As a by-product of this extraction, it also identifies the primitive events which can be observed during the runtime operation of the SBS systems. These events are used by the monitoring manager to check whether the formulas specified in an agreement can be monitored at runtime. They are also used by the assumptions editor of the framework (see below) as primitive constructs for specifying the service guarantee terms that are to be monitored in cases where the service consumers and producers wish to specify these terms using the framework.

Finally, the framework incorporates a *monitoring console* that gives access to the monitoring service to human users. The console incorporates a terms editor that supports the specification of the service guarantee terms of an agreement in the high level syntax of our event calculus–based language, and a *deviation viewer* that displays the deviations from the monitored requirements. The terms editor provides a form-based interface that enables the user to select events extracted from the BPEL process of an SBS system and combine them in order to specify the formulas that define the service guarantee terms of an agreement.

10.3 Specification of Service Level Agreements

10.3.1 Overview of *WS-Agreement*

WS-Agreement is a standard developed by the Global Grid Form for specifying agreements between service providers and service consumers and a protocol for creating and monitoring such agreements at runtime [2]. The objective of a *WS-Agreement* specification is to define the guarantee terms that should be satisfied during the provision of a service. *WS-Agreement* is defined as an XML schema. An agreement drawn using *WS-Agreement* has two sections: the *Context* section and the *Terms* section.

The *Context* section specifies the consumer and the provider of the service that have created the agreement (i.e., the parties of the agreement) and other general properties of the agreement including, e.g., its duration and any links that it may have to other agreements.

The *Terms* section of a *WS-Agreement* specifies the service that the agreement is about and the objectives that the provision of this service should fulfill. This section is divided into two subsections: the *Service Description Terms* and *Service Guarantee Terms*. The service description terms constitute the basic building block of an agreement and define the functionalities of the service that is to be delivered under the agreement. An agreement may contain any number of service description terms. The guarantee terms specify assurances on service quality that need to be monitored and enforced during the provision of a service.

The agreement life cycle that is envisaged by *WS-Agreement* expects that an agreement initiator sends an agreement template to the service consumer. This template is defined by adding a new section to the agreement structure described above, called *Creation Constraints*. This new section contains constraints on possible values of terms for creating the agreement. The consumer fills in the template and sends it back to the initiator as an offer. Subsequently, the initiator notifies the consumer of the acceptance or rejection of the agreement depending on the availability of resources, the service costs, etc. The monitoring of an agreement that has been confirmed is expected to start when at least one of the services which are involved in the agreement is running.

10.3.2 Extensions of *WS-Agreement*

In its original form, *WS-Agreement* does not support the specification of policies determining the deployment context in which the provision of services will be monitored, and who will have responsibility for providing the information that will be necessary for assessing whether the guarantee terms of the agreement are satisfied. Also, it does not specify where the results of monitoring should be reported. This is problematic in cases where the agents who have responsibility for the monitoring of an agreement are expected to actively report deviations from it rather than waiting to be asked if deviations have occurred (i.e., notification of deviations in a push mode). Furthermore, *WS-Agreement* does not specify a language for defining the service description and service guarantee terms of an agreement or an operation protocol that would enable the monitoring of an agreement in the push mode described above. The choice of the language for the specification of the service description and service guarantee terms of an agreement is left to the concrete implementations of the standard as the language for the specification of these terms may need to vary for different domains. Our extensions to *WS-Agreement* address these limitations of the standard.

Specification of the Context of an Agreement

Our first extension to *WS-Agreement* is concerned with the specification of *policies* for monitoring an agreement. A policy, in our proposal, specifies the following:

- The composition process of the SBS system that deploys the services which are the subject of the agreement.
- The source of the runtime information which will enable the monitoring of the agreement.
- The way in which the monitoring of the agreement is to be performed including the mode, regularity, and timing of monitoring.
- The recipient of the results of the monitoring process.

To enable the specification of monitoring *policies*, we have extended *WS-Agreement* by a complex XML type, called *MonitoringPolicyType*. A graphical view of this type is shown in Fig. 10.2. According to *MonitoringPolicyType*, the description of the monitoring policy of an agreement includes the following elements:

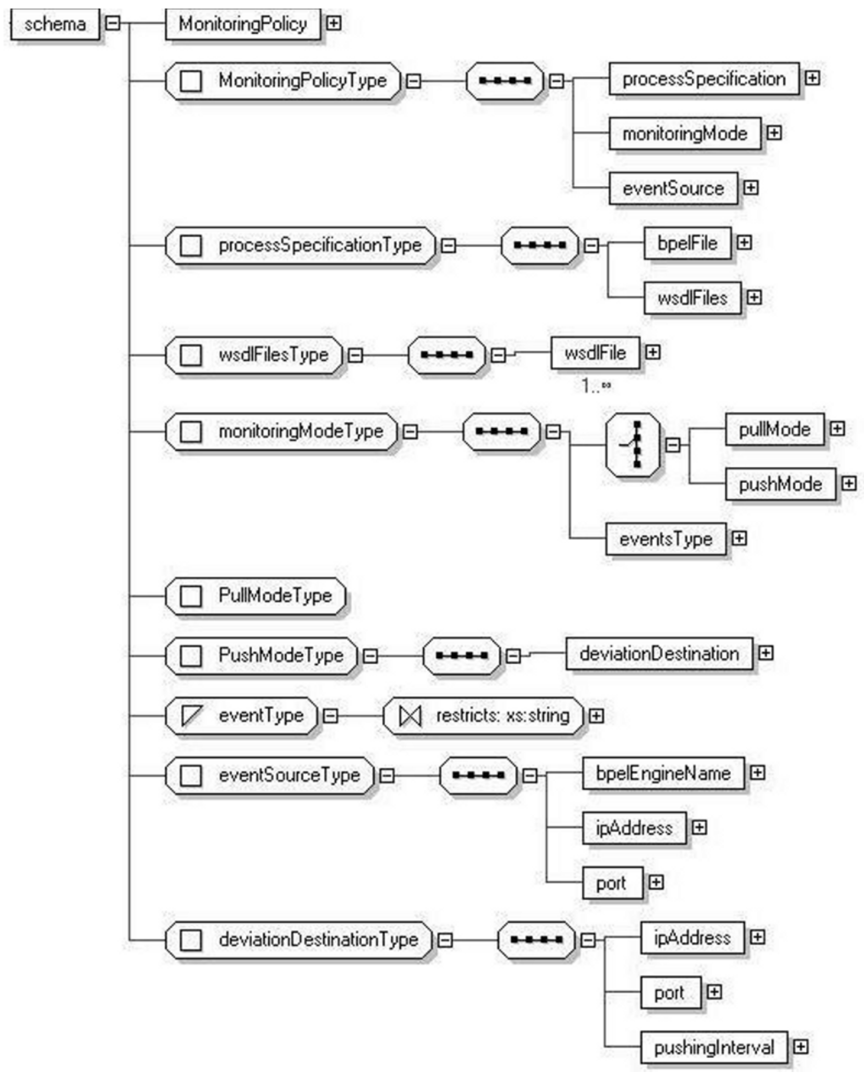


Fig. 10.2. *MonitoringPolicyType* – Specification of agreement monitoring policies

1. *processSpecificationType*. This element is used to identify the BPEL composition process of the SBS system that deploys the service(s) which the agreement is concerned with and the WSDL files of all the services that this process uses but are not regulated by the agreement (called third-party services in the following). The references to the WSDL specifications of third-party services in a monitoring policy is important as the behavior of these services may interfere with the service(s) regulated by

the agreement and, therefore, the guarantee terms in an agreement may need to be conditioned upon the satisfiability of conditions for third-party services. A *processSpecification* includes (a) an element called *bpelFile* that contains a reference to the BPEL file specifying the composition process in the context of which the agreement is to be monitored, and (b) an element called *wsdlFiles* that contains references to the list of the WSDL files that specify the services deployed by the composition process.

2. *monitoringModeType*. Elements of this type are used to specify the way of reporting the results of monitoring an agreement (i.e., the mode of reporting); the kind of events that are used to check whether the agreement's guarantee terms are satisfied; and the source of the events which are used to check an agreement. The results of monitoring may be reported in a *pullMode* or a *pushMode*. In the former mode, the client of the monitor has to check the status of the guarantee terms of the agreement. In the *pushMode*, the monitor reports the detected deviations to the client. When the *pushMode* is selected the destination where deviations should be reported (i.e., the *deviationDestination* element in Fig. 10.2) must also be specified. The specification of a deviation destination includes (a) an element called *ipAddress* that is of type string and is used to specify the IP address of the client where the deviation reports will be sent, (b) an element called port of type *int* which specifies the port in the client where the deviation reports should be sent, and (c) an element called *pushingInterval* of type *long* that defines the time interval between the generation of consecutive deviation reports. The type of the events used in monitoring is specified by the element *eventsType* that is of type *eventType*. Currently, our extension supports two types of events: recorded or derived events. Recorded events are events which are generated during the execution of the composition process of an SBS system. Derived events are events which are generated from recorded events.[1]
3. *eventSourceType*. An event source is described by (a) an element called *bpelEngineName* of type *string* which is used to specify the type of the BPEL engine, i.e., the execution environment of the service centric system, (b) an element called *ipAddress* of type *string* that is used to specify the IP address of the execution environment, and (c) an element called port of type *int* that specifies the port where the runtime events will be sent by the event source.

Overall, a monitoring policy is specified as part of the context of an agreement. The new definition of the context type in *WS-Agreement* that includes the element *monitoringPolicy* which allows the specification and attachment of a monitoring policy to the context of an agreement is shown in Fig. 10.3.

[1] In the monitoring framework that we have developed to support the runtime checking of WS-Agreements, derived events are generated by deduction (see Sect. 10.4.1).

Original Form of *WS-Agreement*	**Extended Form of *WS-Agreement***
<xs:complexType name="AgreementContextType">	<xs:complexType name="AgreementContextType">
<xs:sequence>	<xs:sequence>
<xs:element name="AgreementInitiator"	<xs:element name="AgreementInitiator"
type="xs:anyType" minOccurs="0"/>	type="xs:anyType" minOccurs="0"/>
<xs:element name="AgreementProvider"	<xs:element name="AgreementProvider"
type="xs:anyType" minOccurs="0"/>	type="xs:anyType" minOccurs="0"/>
<xs:element	<xs:element
name="AgreementInitiatorIsServiceConsumer"	name="AgreementInitiatorIsServiceConsumer"
type="xs:boolean" default="true"	type="xs:boolean" default="true"
minOccurs="0"/>	minOccurs="0"/>
<xs:element name="TerminationTime"	<xs:element name="TerminationTime"
type="xs:dateTime" minOccurs="0"/>	type="xs:dateTime" minOccurs="0"/>
<xs:element name="RelatedAgreements"	<xs:element name="RelatedAgreements"
type="wsag:RelatedAgreementListType"	type="wsag:RelatedAgreementListType"
minOccurs="0"/>	minOccurs="0"/>
<xs:any namespace="##other"	<xs:any namespace="##other"
processContents="lax" minOccurs="0"	processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>	maxOccurs="unbounded"/>
	<xs:element name="monitoringPolicy"
	type="MonitoringPolicyType
	minOccurs="0" maxOccurs="1" />
</xs:sequence>	</xs:sequence>
<xs:anyAttribute namespace="##other"/>	<xs:anyAttribute namespace="##other"/>
</xs:complexType>	</xs:complexType>

Fig. 10.3. Extended definition of *AgreementContextType* in *WS-Agreement*

Specification of Service Description and Service Guarantee Terms

WS-Agreement defines a service guarantee term as a term that specifies "an assurance to the service consumer on the service quality and/or resource availability offered by the service provider" (see p. 16 in [2]). In our framework, this definition is refined to include *functional* and *quality of service (QoS) requirements* for the constituent services of an SBS system. Functional and QoS requirements may be associated with

- *qualifying conditions* conditions that must be met for a requirement to be satisfied and enforced if it is not (as defined in [2])
- *assumptions* specifying how the behavior of an SBS system and its constituent services affects the state of the system and therefore the satisfiability of the requirements.

At runtime, the monitor of a *WS-Agreement* checks whether the functional and QoS requirements that are defined as service guarantee terms in the agreement are satisfied. During monitoring, any assumptions that may have been specified for service guarantee terms are also used to generate additional information about the effect of the behavior of an SBS system and its constituent services. The identification of this effect is necessary as it may affect the satisfiability of the service guarantee terms.

Service guarantee terms along with their qualifying conditions and assumptions are specified in our framework using an XML schema that is based on event calculus, called *EC-Assertion*. In the following, we give an overview of event calculus and *EC-Assertion* how it is used in our monitoring framework to specify service guarantee terms, qualifying conditions, and assumptions for *WS-Agreements.*

10.3.3 Overview of Event Calculus

The event calculus (EC) is a first-order temporal formal language that can be used to specify properties of dynamic systems which change over time. Such properties are specified in terms of events and *fluents*.

An event in EC is something that occurs at a specific instance of time (e.g., invocation of an operation) and may change the state of a system. Fluents are conditions regarding the state of a system and are initiated and terminated by events. A fluent may, e.g., signify that a specific system variable has a particular value at a specific instance of time.

The occurrence of an event is represented by the predicate $Happens(e, t, \mathbb{R}(t_1, t_2))$. This predicate signifies that an instantaneous event e occurs at some time t within the time range $\mathbb{R}(t_1, t_2)$. The boundaries of $\mathbb{R}(t_1, t_2)$ can be specified by using either time constants or arithmetic expressions over the time variables of other predicates in an EC formula.

The initiation of a fluent is signified by the EC predicate $Initiates(e, f, t)$ whose meaning is that a fluent f starts to hold after the event e at time t. The termination of a fluent is signified by the EC predicate $Terminates(e, f, t)$ whose meaning is that a fluent f ceases to hold after the event e occurs at time t. An EC formula may also use the predicates $Initially(f)$ and $HoldsAt(f, t)$ to signify that a fluent f holds at the start of the operation of a system and that f holds at time t, respectively.

Special Types of Fluents and Events

EC-Assertion is based on event calculus but uses special types of events and fluents to specify service guarantee terms, and their qualifying conditions and assumptions. More specifically, the fluents in *EC-Assertion* have the form

$$valueOf(fluent\_expression, value\_expression) \tag{10.1}$$

The meaning of the expression 10.1 is that the fluent signified by *fluent_expression* has the value *value_expression.* Furthermore, in this expression:

- *fluent_expression* denotes a typed SBS system variable or a list of such variables. *fluent_expression* may be an
  - *internal variable* that represents a variable of the composition process of an SBS system, or
  - *external variable* that is introduced by the creators of a service level agreement to represent the state of an SBS system at runtime.

If *fluent_expression* has the same name as a variable in the SBS system composition process then it denotes this variable, has the same name with it, and is treated as an internal variable. In all other cases, *fluent_expression* denotes an external variable and its type is determined by the type of *value_expression* as described below.

- *value_expression* is a term that either represents an EC variable or signifies a call to an operation that returns an object of some type. The operation called by *value_expression* may be an internal operation that is provided by the monitoring framework or an operation that is provided by an external web service. If *value_expression* signifies a call to an operation, it can take one of the following two forms:
  (i) $oc : S : O(\_O_{id}, \_P_1, ..., \_P_n)$ that signifies the invocation of an operation O in an external service S.
  (ii) $oc : self : O(\_O_{id}, \_P_1, ..., \_P_n)$ that signifies the invocation of the built-in operation O of the monitor.

  In these forms,
  - $\_O_{id}$ 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 10.1. Note also that a fluent is valid if and only if the type of *fluent_expression* is more general than the type of *value_expression.* If *fluent_expression* is an external variable, the specification of its type is deduced from the type of *value_expression* in a fluent specification. In this case, if *fluent_expression* appears in different fluents that use different *value_expression* terms, the above type validity condition should be satisfied by the types of all the relevant *value_expression* terms. On the other hand, if *fluent_expression* is an internal variable, its type is determined by the specification of the variable in the composition process of the SBS system that it refers to.

The calls to external and internal operations in fluents allow us to deploy complex computations. As shown in Table 10.1, the internal operations of *EC-Assertion*, for instance, can perform various arithmetic operations over numbers and compute statistics of series of numerical values (e.g., compute the average, median, and standard deviation of a series of values), manage lists of primitive values and create new instances of object types which are supported by *EC-Assertion.*

These operations are necessary for checking QoS requirements within the reasoning process of the monitoring framework. The maintenance of lists of primitive data values, for instance, is useful for recording multi-valued fluents (e.g., recording the response times of a service operation). The operation *avg* for instance, which computes the average value of a list of real or integer number, can be used to compute the average response time of a service operation. During a monitoring session, when attempting to unify formulas which include

Table 10.1. Built-in operations for specification and computation of service guarantee terms

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.

such calls, the EC variables which represent the operation parameters are unified first and then the monitor calls the relevant operation. If the operation returns successfully with a return value that is compliant with the type of *fluent_expression*, this value becomes the binding of the term *value_expression.* Otherwise, unification fails. In Sect. 10.4.3, we give examples of monitoring formulas that use built-in operations of the framework. (Table 10.1).

Events in our framework represent exchanges of messages between the composition process of an SBS system and the services coordinated by it. These messages either invoke operations or return results following the execution of an operation and – depending on their sender and receiver – they can be of one of the following types:

1. Service operation invocation events—These events signify the invocation of an operation in one of the partner services of an SBS system by its composition process and are represented by terms of the form

$$ic : S : O(\_O_{id}, \_P_1, ..., \_P_n) \tag{10.2}$$

where O is the name of the invoked operation; S is the name of the service that provides O, $\_O_{id}$ is a variable identifying the exact instance of O's invocation within an execution of the SBS composition process, and

$\_P_1, ..., \_P_n$ are variables indicating the values of the input parameters of O at the time of its invocation.

2. Service operation reply events—These signify the return from the execution of an operation that has been invoked by the composition process of an SBS in one of its partner services and are represented by terms of the form

$$ir : S : O(\_O_{id}) \tag{10.3}$$

where O, S, and $\_O_{id}$, are as defined in (1). Note that the values of the output parameters of such operations (if any) are represented by fluents which are initiated by the above event as discussed below.

3. SBS operation invocation events—These events signify the invocation of an operation in the composition process of an SBS by one of its partner services and are represented by terms of the form

$$rc : S : O(\_O_{id}) \tag{10.4}$$

where S is the service that invokes O, and O, $\_Oid$ are as defined in (1). Note that the values of the input parameters of such operations (if any) are represented by fluents which are initiated by the above event as discussed below.

4. SBS operation reply events—These events signify the reply following the execution of an operation that was invoked by a partner service in the composition process of an SBS and are represented by terms of the form:

$$re : S : O(\_Oid, \_P1, ..., \_Pn) \tag{10.5}$$

where S is the service that invoked O; $\_P_1, ..., \_P_n$ are variables that indicate the values of the output parameters of O at the time of its return, and O, $\_O_{id}$ are as defined in (1).

EC-Assertion uses another type of events which signify the assignment of a value to a variable used in the composition process of an SBS. These are called assignment events and are represented by terms of the form

$$as : aname(\_A_{id}) \tag{10.6}$$

where *aname* is the name of the assignment in the composition process specification, and $\_A_{id}$ 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.

In addition to the EC predicates and event/fluent denoting terms that were discussed above, formulas that express monitorable properties in *EC-Assertion* can use the predicates $<$ and $=$ to express time conditions (the predicate $t1 < t2$ is true if $t1$ is a time instance that occurred before $t2$, and the predicate $t1 = t2$ is true if $t1$ is a time instance that is equal to $t2$) and to compare values of different variables. Also, an EC formula that expresses a monitorable property must specify boundaries for the time ranges $\mathbb{R}(LB, UB)$ which appear in the $Happens$ predicates.

If the variable t in such predicates is existentially quantified, at least one of LB and UB must be specified. These boundaries can be specified by using (i) constant time indicators or (ii) arithmetic expressions of time variables t' which appear in *Happens* predicates of the same formula provided that the latter variables are universally quantified, and that 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 $Happens(e, t, \mathbb{R}(t, t))$. These predicates express instantaneous events. Furthermore, a formula is valid in our framework if the time variables of all the predicates, which include existentially quantified non-time variables, take values in time ranges with fixed boundaries. These restrictions guarantee the ability to check the satisfiability of formulas. Furthermore, a specification of requirements must also be compliant with the standard axioms of event calculus. These axioms are shown in Fig. 10.4.

The axiom $EC1$ in Fig. 10.4 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. 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. The axiom $EC3$ states that a fluent f holds at time t, if it is held at time 0 and has not been terminated between 0 and t. 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 $t2$. 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 $t2$. The axiom $EC6$ states that a fluent f holds at time $t2$, if it held at time $t1$ prior to $t2$ and has not been terminated between $t1$ and $t2$. 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 $t2$

(EC1)	$Clipped(t1,f,t2) \Leftarrow (\exists e,t)\ Happens(e,t,\Re(t1,t2)) \wedge Terminates(e,f,t)$
(EC2)	$Declipped(t1,f,t2) \Leftarrow (\exists e,t)\ Happens(e,t,\Re(t1,t2)) \wedge Initiates(e,f,t)$
(EC3)	$HoldsAt(f,t) \Leftarrow Initially(f) \wedge \neg Clipped(0,f,t)$
(EC4)	$HoldsAt(f,t2) \Leftarrow (\exists e,t)\ Happens(e,t,\Re(t1,t2)) \wedge Initiates(e,f,t) \wedge \neg Clipped(t,f,t2)$
(EC5)	$\neg HoldsAt(f,t2) \Leftarrow (\exists e,t)\ Happens(e,t,\Re(t1,t2)) \wedge Terminates(e,f,t) \wedge \neg Declipped(t,f,t2)$
(EC6)	$HoldsAt(f,t2) \Leftarrow HoldsAt(f, t1) \wedge t1 < t2 \wedge \neg Clipped(t1,f,t2)$
(EC7)	$\neg HoldsAt(f,t2) \Leftarrow \neg HoldsAt(f, t1) \wedge (t1 < t2) \wedge \neg Declipped(t1,f,t2)$
(EC8)	$Happens(e,t,\Re(t1,t2)) \Rightarrow (t1 \leq t2) \wedge (t1 \leq t) \wedge (t \leq t2)$

Fig. 10.4. Axioms of Event Calculus

and f has not been declipped since then. Finally, the axiom $EC8$ states that the time range in a $Happens$ predicate is inclusive of its boundaries.

Examples of Specification of Service Guarantee Terms

In the following, we present examples of functional and QoS guarantee terms that we can specify using *EC-Assertion*. Our examples are based on a simple SBS system, called *Quote Tracker Process (QTP)*, which we have implemented to test the monitoring framework (see [31] for a specification of the BPEL process and the services deployed by this system).

QTP allows a user to get a stock quote in US dollars given a stock symbol from New York Stock Exchange (NYSE) and convert it to some other currency. QTP uses a web service called Stock *Quote Service (SQS)* to get a quote for stocks traded in the New York Stock Exchange using a NYSE symbol. It also uses a second web service called *Currency Exchange Service* (CES) to get the currency exchange rate between US dollars and a target currency, and a third web service, called *Simple Calculator Service (SCS)*, to convert the quote into the target currency. QTP has been implemented as a BPEL process of QTP and uses the services *SQS* and *CES* of *XMethods*. In our implementation, *SCS* is a service that we have developed.

Fig. 10.5 shows specifications of functional and QoS properties for QTP in the high-level logical syntax of *EC-Assertion*.

The formula $F1$ in Fig. 10.5, for instance, specifies a functional requirement for the *CES* service. According to this requirement, any request for the exchange rate between two countries $\_country1$ and $\_country2$ that is sent to CES within a specific time period T should return the same exchange rate. This requirement is specified to ensure the consistency of the information returned by CES. The *EC-Assertion* formula specifies this requirement by stating that the results which are returned by any two invocations of the operation $getRate(\_ID, \_country2, \_country1)$ of the CES service that have happened within a time period $[t1, ..., t1 + T]$ must be the same. The invocations of the operation *getRate* in this case are represented by the predicates $Happens(ic : CES : getRate(\_ID1, \_country2, \_country1), t1, \mathbb{R}(t1, t1))$ and $Happens(ic : CES : getRate(\_ID2, \_country2, \_country1), t3, \mathbb{R}(t1, t1 + T))$ in the formula. The results of the invocations of *getRate* are represented by the initiation of the external fluent variables $Result1$ and $Result2$. The assignment of values to these two fluent variables is expressed by the predicates

- $Initiates(ir : CES : getRate(_ID1), equalTo(Result1, \_result1), t2)$ and
- $Initiates(ir : CES : getRate(ID2), equalTo(Result2, \_result2), t4)$.

The formula $Q1$ in Fig. 10.5 expresses a quality requirement for the CES service of QTP. According to this requirement, the response time of the operation *getRate* of CES should be less than 100 milliseconds (ms). The response time in this formula is measured as the difference between the time of the receipt of the response of CES following the completion of the execution of *getRate* (this is signified by the time variable $t2$ of the predicate $Happens(ir : CES : getRate(\_ID1), t2, \mathbb{R}(t1, t2))$ which represents the return of the operation in the formula) and the time when this operation was invoked in the service (this is signified by the time variable $t1$ of the predicate $Happens(ir : CES : getRate(\_ID1), t2, \mathbb{R}(t1, t2))$ which represents the call of the operation in the formula).

A second quality requirement is expressed by the formula $Q2$ in Fig. 10.5. The requirement that is expressed by this formula is that the average response time of all the invocations of operation *getQuote* of the SQS service which take place in the time range $\mathbb{R}(T1, T2)$ should be less than 100ms. $Q2$ expresses this requirement by requiring the result of the calculation of the average of the values stored in the list $SQS\_get\_Quote\_RT[]$ to be less than 100ms. In $Q2$, $SQS\_get\_Quote\_RT[]$ is specified as an external fluent variable which is updated every time that there is an invocation of *getQuote* followed by a return from the execution of this operation. In these cases, the response time of each invocation is appended to the list of values $SQS\_get\_Quote\_RT[]$. The update of the values of $SQS\_get\_Quote\_RT[]$ is specified by the assumption $A1$ which

```
(F1)   (∀_ID1,_country1,_country2: String) (∀ t1: Time)
       Happens(ic:CES:getRate(_ID1,_country2,_country1),t1, ℜ(t1,t1)) ^  (∃t2:Time) ^
       Happens(ir:CES:getRate(_ID1),t2, ℜ(t1,t2)) ^
       Initiates(ir:CES:getRate(_ID1),valueOf(Result1,_result1),t2) ^ (∃t3:Time) ^
       Happens(ic:CES:getRate(_ID2, _country2, _country1),t3, ℜ(t1,t1+T)) ^ (∃t4:Time) ^
       Happens(ir:CES:getRate(_ID2),t4, ℜ(t3,t4)))
       Initiates(ir:CES:getRate(ID2),valueOf(Result2,_result2),t4) ⇒  _result1 = _result2
(Q1)   (∀_ID,_country1,_country2: String) (∃t1: Time)
       Happens(ic:CES:getRate(_ID,_country1,_country2),t1, ℜ(t1,t1)) ^ (∃t2:Time) ^
       Happens(ir:CES:getRate(_ID),t2, ℜ(t1,t2)) ⇒  oc:self:sub(t2,t1) < 100
(Q2)   (∀ t1: Time)   HoldsAt(valueOf(SQS_get_Quote_RT[],_resTime),t1) ⇒  oc:self:avg(_resTime]) < 100
(A1)   (∀ _ID, _symbol: String) (∃ t1: Time)
       Happens(ic:SQS:getQuote(_ID,_symbol),t1, ℜ(T1,T2)) ^ (∃t2:Time)
       Happens(ir:SQS:getQuote(_ID),t2, ℜ(t1,t2)) ^ HoldsAt(valueOf(SQS_get_Quote_RT[],_resTime),t2)
       ⇒
       Initiates(ir:SQS:getQuote(_ID), valueOf(SQS_get_Quote_RT[], oc:self:append(_resTime,
        oc:self:sub(t2, t1)), t2))
(Q3)   (∀ t1: Time) (t1 = T2+1) ^
       HoldsAt(valueOf(getQuote_responses,_resNumber), t1) ^
       HoldsAt(valueOf(getQuote_fails,_failsNumber), t1)  ⇒
       oc:self:div(_resNumber, oc:self:add(_failsNumber,_resNumber)) > 0.999
(A2)   (∀ _ID, _symbol: String,  t1: Time)
       Happens(ic:SQS:getQuote(_ID,_symbol),t1, ℜ(T1,T2)) ^ (∃t2:Time)
       Happens(ir:SQS:getQuote(_ID),t2, ℜ(t1,t1+500)) ^
        HoldsAt(valueOf(getQuote_responses,_resNumber),t2)  ⇒
       Initiates(ir:SQS:getQuote(_ID), valueOf(getQuote_responses, oc:self:add(_resNumber, 1), t2))
(A3)   (∀ _ID, _symbol: String,  t1: Time)
       Happens(ic:SQS:getQuote(_ID,_symbol),t1, ℜ(T1,T2)) ^ ¬ (∃t2:Time)
       Happens(ir:SQS:getQuote(_ID),t2, ℜ(t1,t1+500)) ^
        HoldsAt(valueOf(getQuote_fails,_failNumber),t2)  ⇒
       Initiates(ir:SQS:getQuote(_ID), valueOf(getQuote_fails, oc:self:add(_failNumber, 1), t2))
```

Fig. 10.5. Functional and Quality of Service requirements for the CES and SQS services of QTP

appends each new response time of *getQuote* to the list of values already in $SQS\_get\_Quote\_RT[]$ (see the fluent initiation predicate $Initiates(ir : SQS : getQuote(\_ID), valueOf(SQS\_get\_Quote\_RT[], oc : self : append(\_resTime, oc : self : sub(t2, t1)), t2))$ in A1).

Formula $Q3$ in Fig. 10.5 expresses a second QoS requirement for the SQS service. According to this formula, the requirement that should be guaranteed for this service is that rate of responses which are received within 500ms after an invocation of the operation *getQuote* of SQS should exceed 99.9%. This requirement is expressed by $Q3$ as a condition over the values of the fluents *getQuote_responses* and $getQuote\_fails$. These two fluents keep the counters of cases where *getQuote* produced a response within 500ms following its invocation and cases where it did not, respectively. The values of these two fluents are updated by deduction from the assumptions $A2$ and $A3$, respectively. More specifically, from $A2$ it can be deduced that the value of the fluent *getQuote_responses* should be increased by one every time that *getQuote* produces a response within 500 ms after its invocation. Similarly, from $A3$ it can be deduced that the value of the fluent $getQuote\_fails$ should be increased by one every time that *getQuote* does not produce a response within 500 ms from its invocation. $Q3$ uses the built-in operations of *EC-Assertion* to calculate the ratio of the values of these two fluents.

As noted earlier, the specification of the formulas in Fig. 10.5 is given in the high-level logic-based syntax of *EC-Assertion*. Our framework supports the transformation of the logic formulas which are specified in this logic-based syntax into an XML-based representation following the schema that defines *EC-Assertion*. This representation is generated by the editor of our framework from the specification of the formula in the high-level EC syntax automatically. Fig. 10.6 shows an extract of the representation of formula $Q2$ in *EC-Assertion*. The highlighted terms in the figure represent the specification of the two *Happens* predicate in the formula. The description of the full syntax of *EC-Assertion* is beyond the scope of this chapter. The specification of it, however, is available in [14] and a graphical representation of the XML schema that defines *EC-Assertion* is given in the appendix of this chapter.

Specification of Service Guarantee Terms

The specification of service guarantee terms using *EC-Assertion* is supported by a refinement of the definition of the sub-elements $QualifyingCondition$ and $ServiceLevelObjective$ in *WS-Agreement*.

The element $QualifyingCondition$ in an agreement is used to specify a precondition that should be satisfied for the enforcement of a service guarantee term [2]. The element $ServiceLevelObjective$ is used to specify a condition that must be met in order to satisfy a service guarantee. The type of both these elements in the original form of *WS-Agreement* is $xs : anyType$

```
<formula forChecking="true" formulaId="Q2">
  <quantification>
   <quantifier>existential</quantifier>
          <timeVariable> <varName>t1</varName> … </timeVariable>
      </quantification>
  <quantification>
   <quantifier>existential</quantifier>
          <timeVariable> <varName>t2</varName> … </timeVariable>
  </quantification>
  <body>
  <predicate negated="false" unconstrained="true">
      <happens>
          <ic_term>
          <operationName>GetRate</operationName> <partnerName>CES</partnerName>
          <id>_ID</id> <varName>_country1</varName> ...
          <varName>_country2</varName></varName>
          </ic_term>
          <timeVar> <varName>t1</varName> … </timeVar>
          <fromTime> <time> <varName>t1</varName> … </time> </fromTime>
          <toTime> <time> <varName>t1</varName> …</time> </toTime>
      </happens>
  </predicate>
  <operator>and</operator>
  <predicate negated="false" unconstrained="false">
  <happens>
      <ir_term>
          <operationName>getRate</operationName> <partnerName>CES</partnerName>
          <id>_ID</id>
      </rc_term>
      <timeVar> <varName>t2</varName> … </timeVar>
      <fromTime> <time> <varName>t1</varName> … </time> </fromTime>
      <toTime> <time> <varName>t2</varName> … </time> </toTime>
  </happens>
  </predicate>
  </body>
  <head>
  <relationalPredicate> <lessThan>
      <operand1> <operationCall> <name>sub</name> <partner>self</partner>
              <variable forMatching="false" …> <varName>t2</varName> … </variable>
              <variable forMatching="false" …> <varName>t1</varName> … </variable>
          </operationCall>
      </operand1>
      <operand2> <constant> <name>Vo</name> <value>1000</value> </constant>          </operand2>
  </lessThan> ...
  </relationalPredicate>
  </head>
  </formula>
```

Fig. 10.6. Extract of the representation of formula $Q2$ in *EC-Assertion*

(Fig. 10.7). In the extended form of *WS-Agreement*, the type of these elements is *ecQualifyingConditionType* and *ecServiceLevelObjectiveType*, respectively.

ecQualifyingCondition is used to specify the precondition of a service guarantee term. *ecQualifyingCondition* is defined as a type with a single sub-element, called formula, of type ecFormula, i.e., the type of EC formulas in *EC-Assertion.*

ecServiceLevelObjectiveType is defined as a type with two sub-elements: one sub-element called *guaranteeFormula* defines the condition that must be met for the service guarantee term to be satisfied and the second sub-element called *assumption* specifies the effects of the behavior on an SBS and its

Extension for specifying *GuaranteeTerms*	
Original Form	**Extended Form**
<xs:complexType name="GuaranteeTermType">	<xs:complexType name="GuaranteeTermType">
<xs:complexContent>	<xs:complexContent>
…	…
<xs:element ref="wsag:QualifyingCondition"	<xs:element ref="wsag:QualifyingCondition"
minOccurs="0"/>	minOccurs="0"/>
<xs:element ref="wsag:ServiceLevelObjective"/>	<xs:element ref="wsag:ServiceLevelObjective"/>
…	…
</xs:complexContent>	</xs:complexContent>
…	…
<xs:element name="GuaranteeTerm"	
type="wsag:GuaranteeTermType"/>	<xs:element name="GuaranteeTerm"
<xs:element name="QualifyingCondition"	type="wsag:GuaranteeTermType"/>
type="xs:anyType"/>	**<xs:element name="QualifyingCondition"**
<xs:element name="ServiceLevelObjective"	**type="xs:ecQualifyingConditionType"/>**
type="xs:anyType"/>	**<xs:element name="ServiceLevelObjective"**
…	**type="xs:ecServiceLevelObjectiveType"/>**
	<xs:complexType
	name="xs:ecQualifyingConditionType">
	<xs:sequence>
	<xs:element name="formula"
	type="ecas:ecFormula"
	minOccurs="1"/>
	</xs:sequence>
	</xs:complexType>
	<xs:complexType
	name="ecServiceLevelObjectiveType">
	<xs:sequence>
	<xs:element name="guaranteeFormula"
	type="ecas:ecFormula" minOccurs="1"
	maxOccurs="1"/>
	<xs:element name="assumption"
	type="ecas:ecFormula" minOccurs="0"
	maxOccurs="unbounded"/>
	</xs:sequence>
	</xs:complexType>

Fig. 10.7. Extensions in *WS-Agreement* for Specifying Service Guarantee Terms

constituent services which affect the satisfiability of a *guaranteeFormula*. The type of both these elements is *ecFormula*, as shown in Fig. 10.7.

10.4 Monitoring Service Level Agreements

10.4.1 Types of Agreement Deviations

A broad distinction that is made by our monitoring framework is related to the type of the events which are used in order to detect deviations from service level guarantee terms. These events may be of two types: (1) Events which have been captured during the operation of the system at runtime or (2) events which are generated from recorded events by deduction. The use of events of these two types also affects the characterisation of deviations from service level agreements in our framework. More specifically, if monitoring is based only on recorded events, it can detect only inconsistencies which are evidenced by violations of specific service guarantee terms 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 behavior, (b) cases of unjustified system behavior, (c) possible inconsistencies evidenced from the expected system behavior, and (d) possible cases of unjustified system behavior.

In the following, we focus only on inconsistencies caused by recorded events and derived events. A description of the other types of inconsistencies that can be detected by our framework is beyond the scope of this chapter and may be found in [36].

10.4.2 Violations of Service Guarantee Terms by the Recorded Behavior of SBS Systems

The *recorded behavior* of an *SBS* system S at time T, $E_R(T)$, is defined as a set of event, and fluent initiation or termination literals of the forms: $Happens(e, t, \mathbb{R}(t, t))$, $Initiates(e, f, t)$, and $Terminates(e, f, t)$ which have been recorded during the operation of S and for which $0 \leq t \leq T$. A violation of a service guarantee term of the form $f : H \Rightarrow B$ is caused by the recorded behavior of a system at time T if the recorded behavior implies the negation of the term, that is if

$$\{E_R(T), EC_a\} \models \neg f$$

where

- $\models$ signifies logical entailment using also the principle of negation as failure, and
- EC_a are the axioms of event calculus.

Assuming the log of the runtime events of QTP shown in Fig. 10.8, the quality requirement $Q1$ that requires the response time of the operation *getRate* of the service CES to be less than 100ms is violated at time $T = 24657$. This is because at this time point an event signifying the response from the execution of this operation that was invoked at $T = 24500$ is received and the time difference between the invocation and the response of the operation is found to be 157ms (see the events L4 and L5 in the event log of Fig. 10.8 which represent the invocation and response of the operation *getRate*, respectively). The identification of the violation is identified since the events L4 and L5 imply the negation of Q1. This is because, following the unification of the variables $t2$ and $t1$ of $Q1$ with the values 24657 and 24500 respectively, the result of the execution of the built-in operation $oc : self : sub(24657, 24500)$ in the formula is not less than 100 as required by $Q1$.

Violations of Service Guarantee Terms by the Expected Behavior of SBS Systems

The second type of deviations that can be detected in our framework are violations of service guarantee terms by the expected behavior of an SBS system. The latter type of behavior includes the set of predicates that can be derived by deductive reasoning from the recorded behavior of a system using the formulas in the behavioral specification B_S of the system and the assumptions A_S that have been specified for it. As defined in [36], a service guarantee term of the form $f : C \Rightarrow A$ is violated with the expected behavior of a system at time T if

$$E_R(T), EC_a, dep(A_S \cup B_S, f) \models \not f$$

where $dep(A_S \cup B_S, f)$ is the set of formulas $g : B \Rightarrow H$ in the assumptions A_S defined for f and the service description terms of the SBS system (BS) which f depends on. In this definition, inter-formula dependencies are defined as follows. A formula f depends on a formula $g : B \Rightarrow H$ if the head H of g has a predicate L that unifies with some predicate K in the body C of f or with some predicate K in the body B'' of another formula g' that f depends on.

The runtime events of Fig. 10.8 and the events that can be derived from them given the assumptions of Fig. 10.5 and the axioms of event calculus violate the QoS requirement Q2 of QTP at $T = 26325$. This is because at this time point, the fluent vector variable $SQS\_get\_Quote\_RT[]$ has two values (50 and 200), the average value of which is not less than 100. The violation in this case is detected using the derived events and recorded events of QTP. More specifically the relevant derived events in this case are the events which update the value of the fluent (vector) variable $SQS\_get\_Quote\_RT[]$. These events are generated by deduction from the assumption $A1$. More specifically, following the events $L1$ and $L2$, $SQS\_get\_Quote\_RT[]$ is deduced by $A1$ to include the value 50 and following the events $L7$ and $L8$ the same fluent variable is deduced to include the value 200 too.

10.4.3 Monitoring Process

In the following, we describe the process by which our monitor checks for violations of service guarantee terms. At runtime, the monitor generates and maintains templates that represent different instantiations of the formulas

```
L1  :   Happens(ic:SQS:getQuote(ID1,SX),23100,ℜ(23100,23100))
L2  :   Happens(ir:SQS:getQuote(ID1),2315,ℜ(23150,23150))
L3  :   Initiates(ir:SQS:getQuote(ID1),valueOf(q,107),23150)
L4  :   Happens(ic:CES:getRate(ID2,US,UK),24500,R(24500, 24500))
L5  :   Happens(ir:CES:getRate(ID2),24657,R(24657, 24657))
L6  :   Initiates(ir:CES:getRate(ID2), valueOf(rate,1.77),24657)
L7  :   Happens(ic:SQS:getQuote(ID3,SY),26125,ℜ(26125,26125))
L8  :   Happens(ir:SQS:getQuote(ID3),26325,ℜ(26325,26325))
L9  :   Initiates(ir:SQS:getQuote(ID3),valueOf(q,54),26325)
L10 :   Happens(ic:CES:getRate(ID4,US,UK),27555,R(27555, 27555))
L11 :   Happens(ir:CES:getRate(ID4),28000,R(28000, 28000)
L12 :   Initiates(ir:CES:getRate(ID4), valueOf(rate,1.77),28000)
```

Fig. 10.8. Runtime events of QTP

that specify the service guarantee terms of an agreement which should be monitored. A template for a formula f stores the following:

- The identifier (Id) of f.
- A list of pairs (i, p) where i indicates a formula that depends on f, and p indicates the predicate that creates the dependency.
- The variable binding (VB) computed for the template (i.e., the set of value bindings of the variables of the formula represented by the template).
- For each predicate p in f
  - The *quantifier* of its time variable (Q) and its signature (SG).
  - The boundaries (LB, UB) of the time range within which p should occur.
  - The *truth value* (TV) of p. TV is defined to be UN if the truth value of the predicate is not known yet, T if the predicate is known to be true, and F if the predicate is known to be false.
  - The *source* (SC) of the evidence for the truth value of p. The value of SC is UN if the truth value has not been established yet, RE if the truth value of the predicate has been established by a recorded event, DE if the truth value of the predicate has been established by a derived event, and NF if the truth value of the predicate has been established by the principle of negation as failure.
  - A *time stamp* (TS) indicating the time in which the truth value of p was established.

The monitor creates two sets of templates for each formula: a set of *deviation templates* which are used to check for violations of the formula, and a set of *derivation templates* which are used to derive predicates from the formula.[2]

Both types of templates are updated by recorded and derived events. Recorded events are captured by the event receiver and stored in the event database of the framework (see Fig. 10.1). These events are processed by the monitor in the exact order of their occurrence and used to update the truth values of predicates in templates. When a new event is taken from the event database, the monitor checks it against all the different templates to establish if the event could be unified with a predicate in the template. In cases where the event can be unified with a predicate in a template and the truth value of the predicate has not been set yet, the template is updated. The form of the update depends on whether the predicate has an existentially or a universally quantified time variable.

More specifically, the truth value of a predicate with an existentially quantified time variable—i.e. a predicate of the form $(\exists t)p(x, t)$ where t is in the range $\mathbb{R}(t1, t2)$—is set to $T(true)$ as soon as the first event e that can be unified with p occurs between $t1$ and $t2$. If no such event occurs at the distinguishable time points within $\mathbb{R}(t1, t2)$, the truth value of p is set to $F(false)$.

[2] Derivation templates are not generated if the mode of monitoring in a monitoring policy is set to recorded events only (see Sect. 10.3).

The absence of events unifiable with p is confirmed as soon as the first event that cannot be unified with p occurs after $t2$. The truth value of a predicate of the form $\not(\exists p)(x, t)$ is established in the opposite way: as soon as an event e that can be unified with p occurs between $t1$ and $t2$, the truth value of p is set to $F(false)$ and if no such events occur at the distinguishable time points between $t1$ and $t2$, the truth value of p is set to $T(true)$.

The truth value of a predicate with a universally quantified time variable—i.e., a predicate of the form $(\forall t)p(x, t)$ where t must be in the range $\mathbb{R}(t1, t2)$—is set to $F(false)$ as soon as an event which is not unifiable with p occurs between $t1$ and $t2$, and to $T(true)$ if all the events that occur at the distinguishable time points between $t1$ and $t2$ can be unified with p. The truth value of predicates of the form $\not(\forall t)p(x, t)$, where t must be in the range $\mathbb{R}(t1, t2)$, is set to $T(true)$ as soon as the first event that is not unifiable with p occurs within the time range $\mathbb{R}(t1, t2)$ and $F(false)$ if all the events at the distinguishable time points between $t1$ and $t2$ can be unified with p. The truth value of predicates of the form $(\forall t)p(x, t)$, where t is unconstrained (i.e., it is defined to be in a range of the form $\mathbb{R}(t, t)$), is set to $T(true)$ as soon as an event that can be unified with the predicate is encountered.

As an example of this process, consider the check of the satisfiability of the formula $Q1$ in Sect. 10.3. Initially, the template of this formula will have the form shown in Fig. 10.9.

Then, when the event $L4$ in the even log of Fig. 10.8 occurs, the monitor detects that it can be unified with the first predicate in the template (i.e., the predicate $Happens(ic : CES : getRate(\_ID, \_country1, \_country2), t1, \mathbb{R}(t1, t1)))$ and creates a new instance of the template in which the event is unified with the predicate. Following the unification, in the new template instance, which is shown in Fig. 10.10, the truth value of the predicate $Happens(ic : CES : getRate(\_ID, \_country1, \_country2), t1, \mathbb{R}(t1, t1))$ is set to T. This is because the time variable $t1$ of the predicate is universally quantified and unconstrained. Also, the source (SC) of this truth value is set to RE (as the value was set due to a recorded event), the timestamp at which the truth value of the predicate was determined is set to 24500 (i.e., the timestamp of the event that was unified with the predicate) and the lower (LB) and upper (UB) time boundaries of the time variable

Template-1							
ID	Q1						
DP							
VB	(_ID,?) (_country1,?) (_country2,?) (1, ?) (t2, ?)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:CES:getRate(_ID,_country1,_country2),t1, $\Re$(t1,t1))	t1	t1	t1	UN	UN
2	∃	**Happens**(ir:CES:getRate(_ID),t2, $\Re$(t1,t2))	t2	t1	t2	UN	UN
3		oc:self:sub(t2,t1) < 100	?	–	–	UN	UN

Fig. 10.9. Template for formula $Q1$

Template-2							
ID	Q1						
DP							
VB	(_ID, ID2) (_country1,US) (_country2, UK) (t1, 24500) (t2, ?)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:CES:getRate(_ID,_country1,_country2),t1, ℜ(t1,t1))	24500	24500	24500	T	RE
2	∃	**Happens**(ir:CES:getRate(_ID),t2, ℜ(t1,t2))	t2	24500	t2	UN	UN
3		oc:self:sub(t2,t1) < 100	?	–	–	UN	UN

Fig. 10.10. Template for formula $Q1$ updated due to the event $L4$

of the predicate are both set to 24500. The update of the template due to the event $L4$ also changes the variable binding of the template. More specifically, the variables $\_ID$, $\_country1$, $\_country2$, and $\_t1$ of the predicate $Happens(ic : CES : getRate(\_ID, \_country1, \_country2), t1, \mathbb{R}(t1, t1))$ are bound to the values ID2, US, UK, and 23500, respectively. Note also that the lower boundary (LB) of $t2$ which is the time variable of the second predicate in the template has been updated so as to be equal to the value bound to $t1$ (i.e., 24500). As a result of the update of the lower bound of $t2$, the truth value of the second predicate in the template will subsequently be updated only by events that happen after $t = 24500$.

When the event $L5$ in the event log of Fig. 10.8 occurs the template of Fig. 10.10 will be updated again. This is because $L5$ can be unified with the second predicate in the template, i.e., the predicate $Happens(ir : CES : getRate(\_ID), t2, \mathbb{R}(t1, t2))$, and has taken place within the time boundaries of this predicate (i.e., after 24500). The result of this update is shown in Fig. 10.11. As shown in this figure, the truth value of the predicate $Happens(ir : CES : getRate(\_ID), t2, \mathbb{R}(t1, t2))$ is set to T, its timestamp is set to 24657, and the source of the truth value of the predicate is set to RE as the event that led to the update was again a recorded event. At this point, the truth value of the only remaining predicate in the template (i.e., the predicate $oc : self : sub(t2, t1) < 100$) can also be computed. This is because $oc : self : sub(t2, t1) < 100$ is not a predicate with a time variable and all

Template-3							
ID	Q1						
DP							
VB	(_ID, ID2) (_country1,US) (_country2, UK) (t1, 24500) (t2, 24657)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:CES:getRate(_ID,_country1,_country2),t1, ℜ(t1,t1))	24500	24500	24500	T	RE
2	∃	**Happens**(ir:CES:getRate(_ID),t2, ℜ(t1,t2))	24657	24500	24657	T	RE
3		oc:self:sub(t2,t1) < 100	–	–	–	UN	UN

Fig. 10.11. Template for formula $Q1$ updated due to the events $L4$ and $L5$

its variables ($t1$ and $t2$) are bound to specific values in the current variable binding of the template.

As the predicate $oc : self : sub(t2, t1) < 100$ refers to an operation (i.e., $oc : self : sub(t2, t1)$), to establish its truth value the monitor must execute this operation. In this case, the monitor will call the operation $oc : self : sub$ using as parameters the values of the variables $t2$ and $t1$ and will substitute the result of this call (i.e., 257) for the operation in the predicate "<" . Following the substitution, the predicate "<" becomes "257 < 100" and consequently its truth value is evaluated to F.

When the truth values of all predicates in a template have been determined, a check for possible formula violations is performed. This check is carried out according to the following rules:

- If the truth value all the predicates in the template is T, the instance of the formula represented by the template is satisfied.
- If the truth value of all the predicates in the body of the template is T and the truth value of at least one predicate in the head is F and there is no predicate in the template whose source is a derived event (i.e., DE), the instance of the formula represented by the template is inconsistent with the recorded behavior of the system.
- If the truth value of all the predicates in the body of the template is $True$ and the truth value of at least one predicate in the head of the template is $False$ and the source of at least one predicate in the template is a derived event, the formula is inconsistent with the expected behavior of the system.

The template checking process will be triggered following the update of the truth value of the predicate $oc : self : sub(t2, t1) < 100$ in the template of Fig. 10.11. This process will establish that the specific instance of the formula $Q1$ that is expressed by the template has been violated and since the events that have been taken into account in order to establish the truth values of the predicates in the formula are all recorded events, the detected violation is classified as a violation due to recorded behavior.

The monitor also generates derived events by deduction from event derivation templates. More specifically, if in an event derivation template the truth-value of all the predicates in the body of the template is T and there is a predicate p in the head of the template that has an unknown truth value but whose variables are bound to specific values in the variable binding of the template, the truth value of p is set to T and the monitor generates a derived event as a copy of the bound form of p.

Derived events are used to update derivation templates in order to derive further events and to detect deviations. In the update process for derived events, the truth value of a predicate of the form $(\exists t)p(x, t)(\neg(\exists t)p(x, t))$, where t is in the range $\mathbb{R}(t1, t2)$ in a template, is set to $T(F)$ if there is a not-negated derived event e that can be unified with p and the range $\mathbb{R}(t1', t2')$ of e is within $\mathbb{R}(t1, t2)$. The truth value of a predicate of the form $(\forall t)p(x, t)$

(where t is in the range $\mathbb{R}(t1, t2)$) with a yet unknown truth value is set to T if there is a derived event e that can be unified with p and the range $\mathbb{R}(t1, t2)$ is within the range $\mathbb{R}(t1', t2')$ of e. The truth value of a predicate in a template of the form $\neg(\forall t)p(x, t)$ (where t is in the range $\mathbb{R}(t1, t2)$) with a yet unknown truth value is set to T if there is a derived negated event e that can be unified with p and the range $\mathbb{R}(t1', t2')$ of e is within the range $\mathbb{R}(t1, t2)$.

According to this process, the derivation template *Template-4* shown in Fig. 10.12 will be created from formula $A1$ at $T = 23150$. This template will be created from an uninstantiated template of $A1$ following the events $L1$ and $L2$ in Fig. 10.8. More specifically, the truth value of the predicate $Happens(ic : SQS : getQuote(\_ID, \_symbol), t1, \mathbb{R}(T1, T2))$ in *Template-4* will be set to T due to the event $L1$ and the truth value of the predicate $Happens(ir : SQS : getQuote(\_ID), t2, \mathbb{R}(t1, t2))$ will be set to T due the event $L2$ at $t = 23150$. At this time point, the truth value of the predicate $HoldsAt(valueOf(SQS\_get\_Quote\_RT[], \_resTime), t2)$ in the template can also be derived from axiom $EC3$ of event calculus and the predicate $Initially(valueOf(SQS\_get\_Quote\_RT[], []))$ which represents the initial set of the response times of the operation *get_Quote*. Thus, since all the predicates in the body of the template *Template-4* are true, the monitor will use *Template-4* to deduce the truth value of the predicate $Initiates(ir : SQS : getQuote(\_ID), valueOf(SQS\_get\_Quote\_RT[], oc : self : append(\_resTime, oc : self : sub(t2, t1)), t2))$ in the head of the template at $T = 23150$ deriving the following bounded form of this predicate: $Initiates(ir : SQS : getQuote(\_ID), valueOf(SQS\_get\_Quote\_RT[], [50]), 23150))$. This bounded form is derived by first evaluating the term $oc : self : sub(23150, 23100)$, substituting its result (i.e., the value 50) into the term $oc : self : append(\_resTime, oc : self : sub(t2, t1))$ and finally evaluating the latter term. The result of the latter evaluation is the list of values: [50]. This list is substituted for the term $oc : self : append(\_resTime, oc : self : sub(t2, t1))$ in the predicate.

Template-4							
ID	A1						
DP	Q2						
VB	(_ID, ID1) (_symbol, SX) (_resTime, [])						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**Happens**(ic:SQS:getQuote(_ID,_symbol),t1, ℜ(T1,T2))	23100	23100	23100	T	RE
2	∃	**Happens**(ir:SQS:getQuote(_ID),t2, ℜ(t1,t2))	23150	23150	23150	T	RE
3	∃	**HoldsAt**(valueOf(SQS_get_Quote_RT[],_resTime) , t2)	23150	23150	23150	T	DE
4	∃	**Initiates**(ir:SQS:getQuote(_ID), valueOf(SQS_get_Quote_RT[], oc:self:append(_resTime, oc:self:sub(t2, t1)), t2))	23150	23150	23150	UN	UN

Fig. 10.12. Template for formula $A1$

Template-5							
ID	Q2						
DP							
VB	(_resTime, [50])						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**TV**	**SC**
1	∀	**HoldsAt**(valueOf(SQS_get_Quote_RT[], _resTime),t1)	23150	23150	23150	T	DE
2		oc:self:avg(_resTime]) < 100	23150	23150	23150	T	DE

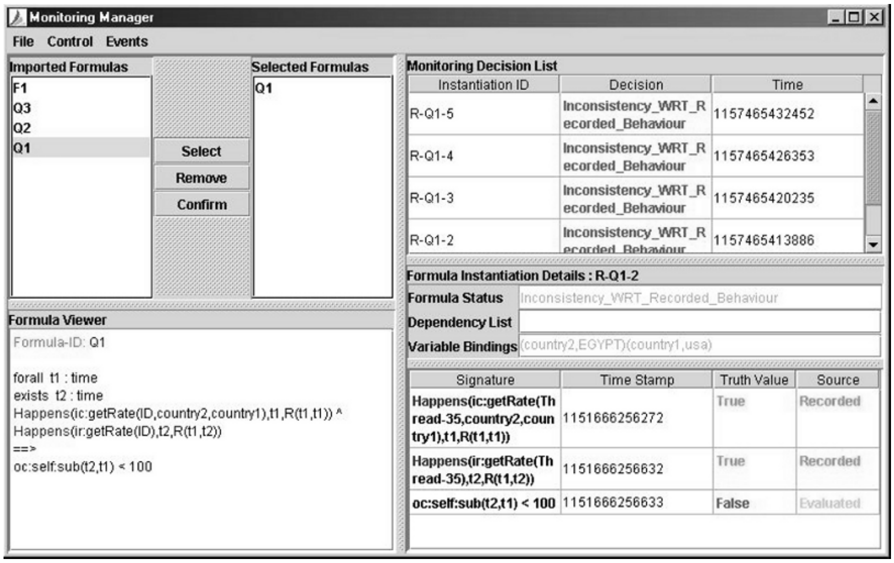
Fig. 10.13. Template for formula $Q2$

Subsequently, from the derived predicate $Initiates(ir : SQS : getQuote(\text{-}\_ID), valueOf(SQS\_get\_Quote\_RT[], [50]), 23150))$ and the axiom $EC3$ of event calculus the predicate $HoldsAt(valueOf(SQS\_get\_Quote\_RT[], [50]),\text{-}23150)$ can also be derived. The latter predicate can then be used to update a deviation template for the formula $Q2$. The template that results from this update and the evaluation of the predicate $oc : self : avg(\_resTime]) < 100$ in the formula is shown in Fig. 10.13. Thus, at $T = 23150$ the formula $Q2$ is satisfied.

10.5 Implementation

Our framework has been implemented by a prototype written in Java^{T}M. This prototype realizes the architecture of the framework that we discussed in Sect. 10.2 and can monitor SBS systems whose composition process is specified in BPEL. In the prototype, we have used the *bpws4j* BPEL process execution engine [6]. This engine uses *log4j* [26] to generate logs of the events during the execution of the composition process of an SBS. This event log is fed into our framework in order to provide the runtime information that is necessary for monitoring. The output of *log4j* is analyzed by the event receiver of the prototype in order to extract the events which are taken into account during the monitoring process.

Figure 10.14 shows a snapshot of the monitoring console of the prototype. This snapshot was taken during a session of monitoring an implementation of the SBS system that we described in Sect. 10.3. The upper left panel of the monitoring console shows the formulas that express the service guarantee terms in a *WS-Agreement* for this system. Using the console, the user of our framework can select one or more of the formulas in an agreement to monitor. Once selected, a formula appears in the lower left panel of the console. In Fig. 10.14, the formula $Q1$ has been selected for monitoring and its EC specification is shown in the lower left panel. When monitoring is activated, the cases which violate and satisfy the selected formulas are shown in the monitoring console (see upper right panel of the console in Fig. 10.14). The user can select any of these cases in order to see the exact instantiation of the formula (template) that underpins the case. This instantiation includes



Imported Formulas: F1, Q3, Q2, Q1

Selected Formulas: Q1

Formula Viewer

```
Formula-ID: Q1

forall t1 : time
exists t2 : time
Happens(ic:getRate(ID,country2,country1),t1,R(t1,t1)) ^
Happens(ir:getRate(ID),t2,R(t1,t2))
==>
oc:self:sub(t2,t1) < 100
```

Monitoring Decision List

Instantiation ID	Decision	Time
R-Q1-5	Inconsistency_WRT_Recorded_Behaviour	1157465432452
R-Q1-4	Inconsistency_WRT_Recorded_Behaviour	1157465426353
R-Q1-3	Inconsistency_WRT_Recorded_Behaviour	1157465420235
R-Q1-2	Inconsistency_WRT_Recorded_Behaviour	1157465413886

Formula Instantiation Details : R-Q1-2

Formula Status	Inconsistency_WRT_Recorded_Behaviour
Dependency List	
Variable Bindings	(country2,EGYPT)(country1,usa)

Signature	Time Stamp	Truth Value	Source
Happens(ic:getRate(Thread-35,country2,country1),t1,R(t1,t1))	1151666256272	True	Recorded
Happens(ir:getRate(Thread-35),t2,R(t1,t2))	1151666256632	True	Recorded
oc:self:sub(t2,t1) < 100	1151666256633	False	Evaluated

Fig. 10.14. Snapshot of Monitoring

the events that have been unified with the different predicates in the formula, the source, and timestamp of each of these events and the truth values of the predicates that the events have been unified with.

Figure 10.14 shows four violations of the formula $Q1$ which, as discussed in Sect. 10.3, requires that the response time of the operation *getRate* of the service Currency Exchange Service should always be less than 100ms. These violations are identified as *R-Q1-5*, *R-Q1-4*, *R-Q1-3*, and *R-Q1-2* in Fig. 10.5. The user can select any of these violations to see the details of the events that have caused it. The violation selected in the figure corresponds to the instance *R-Q1-2* of the formula. As shown in the lower right panel of the figure, this violation has occurred since there was a call of the operation *getRate* (i.e., the event $ic : getRate(Thread - 35, country2, country1), tR(t, t))$ at T=1151666256272 and the response to this call (i.e., the event $ir : getRate(Thread - 35), tR(t, t))$ occurred $T = 1151666256272$. Following the unification of these two events with the $Happens$ predicates in the body of $Q1$, the monitor executed the operation $oc : self.sub(t2, t1)$ in the head of the formula and as the result of this operation was 360 ms the truth value of the "<" predicate in the head of $Q1$ was evaluated to F and the whole formula instance was violated.

The current implementation of the framework does not support the checking of past EC formulas (a past EC formula is a formula in which a predicate p that has a time variable which is constrained by the time variable of an unconstrained predicate q[3] must occur before q). It also assumes that all the non-time variables in formulas are universally quantified and does not support the invocation of external operations in order to perform complex computations during the reasoning process (only invocations of built-in operations are supported by the current implementation). Furthermore, based on our implementation, we found that the impact of generating the events used for monitoring onto the performance of the monitored system was about 18% (i.e., the extra time that it takes from the BPEL engine to create an event log). Currently, we are developing event captors which instead of using *log4j* to generate events capture SOAP messages which are sent to and from the BPEL execution engine and transform these messages into EC events before sending them to the monitor. The impact of this alternative event capturing solution on both the performance of the SBS system that is being monitored and the monitoring process itself is to be evaluated.

10.6 Evaluation

10.6.1 Experimental Set Up

To evaluate the monitoring framework, we have carried out a series of experiments. The objectives of these experiments were as follows:

1. To measure the efficiency of monitoring in terms of the average time that it takes to detect a formula violation from the time that it occurred.
2. To establish 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 non-time variables used in the formulas.

In our experiments, we used an implementation of a *Car Rental System* (CRS) that is described in [12]. This system acts as a broker to car rental companies enabling the hire of cars from different car parks. The system has been implemented by a BPEL composition process that coordinates interactions with different web services including (a) services which provide access to information about the car fleet of different companies and the availability of cars at different car parks and can make car rental reservations and (b) services which operate as drivers of sensors installed at different car parks, tracking car entries and departures.

Our experiments were based on simulations of the BPEL process of CRS. In these simulations, we initially extracted the set of all the possible distinct execution paths of the BPEL process of CRS and expressed them as formulas in *EC-Assertion*. Then we generated different execution paths of the process by selecting different formulas from the extracted formulas set and generating

[3] A predicate p is constrained (unconstrained) if the range of its time variable is (not) defined in terms of the values of other time variables.

randomly events from each selected formula. The random generation of events from a formula was controlled by the following parameters:

1. The size of the domain of the non-time variables in the formula (i.e., the number of the distinct possible values of each variable).
2. The distribution of the values of the constrained time variables within the formula.
3. The distribution of the time that elapses between the initial events of each consecutively selected formulas or, equivalently, the distribution of the values of the unconstrained time variables in the formulas.

Our experimental design covered two different factors that could affect the performance of the monitoring process, namely:

- The *frequency of events*—To explore this factor, we ran simulations of high and moderate event frequency. These two categories of event frequency were controlled by the distribution of the time between the starting events of two consecutively selected formulas in the simulations. In high event frequency (HEF) simulations, the difference between the timestamps of the starting events of two consecutively selected formulas had a *normal distribution* with an average of 3 seconds and a standard deviation of 0.8 seconds. In moderate event frequency (MEF) simulations, the difference between the timestamps of the starting events of two consecutively selected formulas had a *normal distribution* with an average of 10 seconds and a standard deviation of 0.8 seconds. In both simulation categories, the timestamps of the constrained predicates in the formulas were distributed according to the uniform distribution within the range defined by their boundaries. Based on these parameters, in HEF simulations we generated 30,000 events per hour on average and in MEF simulations we generated 9,000 events per hour on average.
- The *size of the domain*—To explore this factor, we ran simulations using large and small domain sizes denoted as LD and SD, respectively. In our case study, we had three different domains for non-time variables, namely customers, cars, and car parks. In LD simulations, we used sets of 200 customers, 80 cars, and 12 car parks. In SD simulations, we used sets of 50 customers, 20 cars, and 3 car parks (i.e., domains whose size was 1/4 of the size of the respective LD domain).

In total, we performed four different experiments in which we monitored four functional requirements for CRS with an average of seven predicates per formula. The experiments were categorized with respect to the previous two differentiation factors as shown in Table 10.2. In each of these experiments, we generated 30,000 events using the simulator of our framework (see Fig. 10.1) and the parameter values that were described above and fed them into the monitor which carried out the monitoring process. The number of events that were used in our experiments corresponded to about 1 hour of operation of

Table 10.2. Classification of experiments

	MEF	HEF
DL	Exp 1	Exp 3
DS	Exp 2	Exp 4

CRS in the case of HEF experiments and 3.3 hours of operation in the case of MEF experiments.

In each experiment, we computed the in making a decision about possible violations of a formula, called d-delay. *d-delay* was measured as:

$$d - delay = \sum_{j=1,\ldots,N} d_j / N$$

In this formula, N is the number of the formula templates for which a decision was made during the experiment and d_j is the delay in making the decision for a formula template j. d_j was computed as

$$d_j = T_E^{F_j} - max_{i \in F_j}(t_i^{e(d)}) \; if \; T_E^{F_j} - max_{i \in F_j}(t_i^{e(d)}) \geq 0 \tag{10.7}$$

$$d_j = T_E^{F_j} - max_{i \in F_j}(t_i^{M}) \; if \; T_E^{F_j} - max_{i \in F_j}(t_i^{e(d)}) < 0 \tag{10.8}$$

where

- t_i^e is the time of occurrence of an event i as generated by the simulator
- T_s^m is the starting time of the monitor
- T_c^m is the current time of the monitor
- $t_i^{e(d)}$ is the time of recording an event i in the monitor's 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 first event that is generated by the simulator. $t_i^{e(d)}$ is the relative time of the occurrence of the event i after the occurrence of the first event that is processed with the monitor and which is assumed to coincide with the starting time of the monitoring process (i.e., T_s^m)
- t_i^M is the time when the monitor retrieves an event i from its database to process it
- $T_S^{F_j}$ is the starting time of the decision procedure that the monitor executes in order to check for violations after the truth values of the predicates in the template j for a formula F have been established
- $T_E^{F_j}$ is the completion time of the decision procedure that the monitor executes in order to check for violations after the truth values of the predicates in the template j for a formula F have been established
- i ranges over the events used to establish the truth values of the predicates in a template F_j.

Formula 10.7 above is used to compute the delay in making a decision about a template in cases where the monitor starts checking this template after the occurrence of all the events that were used to instantiate and set the truth values of the predicates in the template. Formula 10.8 is used in cases where the monitor is capable of checking a template before the last event that was used to instantiate one of its predicates really occurred (this case was only possible due to the use of simulations in which the time of the real occurrence of an event could be after its generation by the simulator and its transmission to the monitor).

Furthermore, for each of the four experiments we produced two sets of results. The first set recorded the average delay in monitoring sessions using only the events generated by the simulator (i.e., recorded events). The second set included the average delay in monitoring sessions using both events generated by the simulator and additional events that were derived from them using the monitored requirement formulas and assumptions (i.e., both recorded and derived events).

10.6.2 Results

Tables 10.3 and 10.4 show the average *d-delay* in making decisions about the satisfiability of monitored formulas that was measured in our experiments. The average delay measures in Table 10.3 refer to monitoring sessions where only the recorded events (i.e., the events generated by the simulator) were taken into account. The average delay measures in Table 10.4 refer to monitoring sessions where both the recorded events and events that could be derived from them and assumptions by deduction were taken into account. The measures appearing in both tables are in seconds.

The experimental results shown in Tables 10.3 and 10.4 demonstrate that the frequency of events had a significant impact on the performance

Table 10.3. *d-delay* with recorded events

	Exp 1	Exp 2	Exp 3	Exp 4
# Events	Avg *d-delay*	Avg *d-delay*	Avg *d-delay*	Avg *d-delay*
2500	0.06	0.62	0.03	0.04
5000	0.14	0.13	15.34	15.21
7500	0.21	0.20	185.95	210.22
10000	0.28	0.28	535.56	585.42
12500	0.36	0.36	1184.38	1195.02
15000	0.43	0.43	1896.70	2010.52
17500	105.56	66.29	2781.94	3034.48
20000	620.06	531.39	4121.51	4086.69
22500	1598.23	1397.85	5476.30	5590.25
25000	2993.66	2666.99	7003.27	7137.67
27500	4643.43	4343.57	9055.78	8838.63
30000	6493.96	6150.47	10671.60	11148.4

Table 10.4. *d-delay* with mixed (recorded and derived) events

	Exp 1	Exp 2	Exp 3	Exp 4
# Events	Avg *d-delay*	Avg *d-delay*	Avg *d-delay*	Avg *d-delay*
2500	49.47	44.70	61.04	68.90
5000	190.65	166.84	451.17	442.47
7500	380.23	342.41	1310.37	1380.84
10000	628.59	569.36	2650.79	2789.45
12500	1059.32	989.13	4420.99	4675.73
15000	1845.82	1744.56	6592.42	6971.83
17500	3205.74	3059.82	9435.23	9970.17
20000	5130.91	4926.66	12992.60	13366.00
22500	7624.29	7266.17	17063.41	17619.07
25000	10607.41	10134.35	21539.77	22325.48
27500	13976.86	13524.91	26647.47	27339.53
30000	17923.68	17452.27	31843.39	33171.97

of the monitor. The average decision delay increased linearly up to a certain number of events and then it increased exponentially. In high event frequency experiments, the exponential rise occurred earlier than in the moderate event frequency (MEF) experiments. More specifically in MEF experiments where only recorded events were used (i.e., *Exp*1 and *Exp*2), the exponential rise of *d-delay* started in the range of 17,500–20,000 events as shown in Table 10.3 (i.e., the equivalent of about 1.9 hours of operation of CRS), whereas in the recorded event HEF-experiments (i.e., *Exp*3 and *Exp*4) the exponential rise of *d-delay* started in the range of 5,000–7,500 events (i.e., after about 0.2 hours of operation of CRS). The same phenomenon was observed for the experiments where we used both recorded and derived events as shown in Table 10.4.

Our experiments also showed that the size of the domains of the non-time variables had no significant effect on the performance of the monitor. This is evident from comparing the *d-delay* between *Exp*1 and *Exp*2 and between *Exp*3 and *Exp*4 both in the case of experiments where only recorded events were used (see Table 10.3) and in the case of experiments where both recorded and derived events were used (see Table 10.4). The reason for the absence of any effect of this factor is likely to have been due to the fact that in our experiments the monitored formulas had predicates which had a small number of shared variables. Thus, increments in the size of the domains of these variables did not lead to a combinatorial proliferation in the number of the templates (instances) of the formulas during the monitoring process.

Our experiments also demonstrated that the use of mixed events had a significant effect on *d-delay*. Table 10.5 shows the ratio of the average *d-delay* of mixed events over the average *d-delay* of recorded only events that was measured in different experiments after processing 10,000, 20,000, and 30,000 events. In both MEF and HEF experiments, this ratio decreased as more events were being processed going down to less than 3 at 30,000 events. This

Table 10.5. Mixed vs recorded events *d-delay* ratio

# Events	Exp 1	Exp 2	Exp 3	Exp 4
10,000	2197.85	2033.42	4.95	4.76
20,000	8.27	9.27	3.15	3.27
30,000	2.76	2.84	2.98	2.98

was due to the fact that at 30,000 events the monitor had been saturated with events and substantial delays were being observed for recorded events alone anyway. At the initial states of monitoring (10,000 events), however, the use of mixed events (i.e., deduction in the monitoring process) caused a substantial difference in *d-delay* which in the case of HEF experiments was almost 5-fold and in the case of MEF experiments (*Exp*1 and *Exp*2) reached a ratio of more than 2,000. The reason for the latter ratio was that monitoring based on recorded events only in the case of MEP experiments was very efficient up to 10,000 events (*d-delay* in this case was less than 0.3 seconds). These results clearly demonstrate, as expected, that the use of deduction affects substantially the efficiency and, therefore, the applicability of the monitoring process.

Also, our experimental results have demonstrated that the average delay in the detection of a formula deviation was substantial after some time. This confirmed the results of a smaller scale experimentation that have been reported in [29, 36]. The observed decision delays suggest that monitoring can be deployed only for 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).

10.7 Related Work

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 recognized in industry and academia [2, 5, 23]. As a result of this recognition, several standards and approaches have emerged, in addition to *WS-Agreement* that we overviewed in Sect. 10.3.1.

WSLA (Web Service Level Agreement) is another framework that can be used to specify a service level agreement between a service provider and service consumer and the obligations of the two parties [23, 28]. This framework provides an XML-based language for specifying quality objectives only (e.g., service performance and throughput) without covering functional requirements. A web service level agreement is agreed and signed by both parties (known as signatory parties) through negotiation. Signatory parties may monitor directly the agreement or employ one or more third parties (known as *supporting parties*) to monitor it. The *WS-Agreement*-based framework that we have described in this chapter provides support for the specification of the entire range of quality properties that can be specified in *WSLA*.

WS-Policy is a W3C standard that provides an XML-based language for expressing the capabilities, requirements, and general characteristics of entities in an SBS system [3]. *WS-Policy* focuses on the provision of operators for combining assertions that specify the above characteristics into policies and the specification of qualifiers indicating the circumstances under which an assertion has to be met. However, it does not provide the equivalent of a full logic–based language that would be required in order to express arbitrary logical conditions regarding the capabilities and requirements of services in a service-based system and does not support the specification of assertions that should hold over specific periods of time. Thus, it does not have the expressive power that is necessary in order to express the entire range of service guarantee terms that might be required as part of a *WS-Agreement*. It would not, for instance, be possible to express the functional requirement F1 in Fig. 10.5 using *WS-Policy*.

Baresi et al. [5] have developed a monitoring tool that supports the monitoring of assertions inserted into the composition process of an SBS system. This work also assumes composition processes specified in BPEL. An assertion is checked by a call to an external service and the execution of the composition process waits until the monitor returns the result of the check. Then, the execution of the composition process may continue or be aborted with the raise of an exception depending on whether the assertion has been violated. The main difference between the work of Baresi et al. and our framework is that the latter cannot perform preventive monitoring in which the violation of a certain property can block the execution of a system operation as [5]. However, our approach is not intrusive to the normal operation of an SBS system and, therefore, the monitoring that it can perform does not affect the performance of the monitored system. Furthermore, our approach makes it possible to monitor more than one service guarantee terms not in isolation (as in [5]) but jointly and complex service guarantee terms which involve conditions over time.

Baresi et al. [4] have also used the *WS-Policy* framework to support the monitoring of security properties for BPEL processes. In this approach the constraints to be monitored are expressed in *WS-Policy* and *WS-PolicyAttachment* is used to attach the policy to a particular context of the BPEL process. Monitoring is performed using the approach described in [5], i.e., given the specification of the constraints to be monitored in *WS-Policy* and *WS-PolicyAttachment*, a process weaver instruments the BPEL process to make it invoke an external service at runtime that checks the relevant constraints.

Another approach for monitoring SBS systems has been developed by Robinson [33]. In this approach, requirements are expressed in KAOS and analyzed to identify obstacles for them (i.e., conditions under which the requirements can be violated). Obstacles are identified by negating a requirement formula R and then identifying all the primitive events that can imply $\neg R$ through a regressive analysis of formulas that $\neg R$ depends on. If an obstacle is observable (i.e., it corresponds to a pattern of events that can be observed at runtime), it is assigned to an agent for monitoring it. At runtime, an event adaptor translates web service requests and replies expressed as SOAP messages into events and a broadcaster forwards these events to the obstacle monitoring agents, which are registered as event listeners to the broadcaster.

Farrell et al. [17] have developed an ontology to capture aspects of service level agreements agreed between service provider and consumer. This work is concerned with the monitoring of properties related to computational resources used by services such as computational power, storage, and network bandwidth. A service level agreement in this approach is specified in terms of an ontology that includes (i) contract management norms defining the effects of contract events on the contract state, (ii) obligation norms that define the actions a party has to perform in case of violation/fulfillment of contract management norms, and (iii) privilege norms that define non-contractual actions that the parties of an agreement are permitted to perform. Contracts in [17] are specified in an XML-based language called CTXML that has a semantics grounded on event calculus. Their framework is supported by a query execution engine that checks whether a CTXML contract is satisfied at runtime. The contract deviations that can be detected in the framework of Farrell et al. are similar to the inconsistencies caused by the recorded behavior of a system in our framework.

Ludwig et al. [28] have developed architecture for a middleware that can be used to create and monitor *WS-Agreement*s, called *Cremona*. *Cremona* has a Java library that implements the protocol for creating service level agreements as defined by *WS-Agreement*. It also proposes the use of monitors that can check the status of the service guarantee terms in an agreement. These monitors are seen in [28] as domain-specific components that can gather primitive information from the systems that provide and/or use a service and use it to evaluate the status of service guarantee terms. As no further information is available to us regarding the implementation of such monitors, we are unable to compare them with the monitoring framework described in this chapter.

Runtime requirements monitoring has been the focus of different strands of requirements engineering research since the late 1990s. Most of the existing techniques (e.g., [19]) express requirements in the KAOS framework [13] as high-level goals that must be achieved by a system. These goals are mapped onto events that must be monitored at runtime. Typically, the existing approaches assume that the events to be monitored are generated by special statements, which must be inserted in the code of a system for this purpose (i.e., *instrumentation*) [32]. Note, however, that instrumentation cannot be always applied to SBS systems since typically SBS system providers are not the owners of the services deployed by the system.

The acquisition of information about the environment of a system during monitoring is even more difficult and most of the approaches do not address this problem. As a solution to this problem, which is prominent in highly dynamic settings (e.g., in mobile computing), Capra et al. [9] have suggested the use of reflective middleware. Such middleware could maintain metadata about an application and its execution context and give dynamic access to this information upon request. In this approach, applications can influence the middleware behavior by changing their own profile based on the reflected information provided by the middleware. The reflective approach is also used in the monitoring framework proposed in [11, 15, 20].

Recently, there has also been work that is concerned with the runtime verification of program behavior [7, 8, 22, 24]. Work in this area focuses on the development of framework for emitting and tracing program events during the execution of a program and verifying them at runtime against properties specified in some formal language, typically a variant of temporal logic. Events normally correspond to change values of program variables at the start or end of method executions. Work in this area focuses on the runtime verification of Java programs and the deployed runtime events are generated either by instrumentation [22, 24] or by using Java debugger interface [7, 8]. These approaches are more close to debugging or perpetual testing of Java programs rather than monitoring high-level user requirements.

10.8 Conclusions and Directions for Future Work

In this chapter, we presented a framework that we have developed to support the monitoring of service level agreements. The agreements that can be monitored are expressed in an extension of *WS-Agreement* that we have described in this chapter. The main characteristic of this extension is that it uses an event calculus–based language, called *EC-Assertion*, for the specification of the service guarantee terms that constitute the core of a service level agreement and specify the conditions regulating the provision of services that should be monitored at runtime. The use of *EC-Assertion* for specifying service guarantee terms provides a well-defined semantics to the specification of such terms and a formal reasoning framework for assessing their satisfiability.

EC-Assertion enables the specification of complex service guarantee terms using full first-order logic formulas as well as conditions about time which are necessary for the specification of not only behavioral but also quality of service guarantees. It also enables the use of well-understood reasoning procedures for the assessment of the satisfiability of service level agreements by our framework. In addition to these characteristics, it should be noted that *EC-Assertion* defines special events and operations which can be used in event calculus formulas to enable the specification of complex service guarantee terms. The use of internal and external operations in formulas enables the delegation of computations of complex data functions which are often required for the specification of service guarantee terms to computational entities outside the main reasoning engine which checks the satisfiability of the terms.

The monitoring framework that supports the proposed extension of *WS-Agreement* has been evaluated in a series of experiments that we reported in this chapter. These experiments have shown that the adoption of non intrusive monitoring approach of our framework introduces some delay in the detection of the deviations for an agreement but does not affect the performance of the system which is being monitored significantly.

Beyond performance, it should be noted that, although our framework is expressive enough to support a wide spectrum of monitorable service guarantee terms, we appreciate that the use of *EC-Assertion* for the specification of such terms may be difficult for users who are not familiar with formal languages. To address this point, we are investigating the development of patterns that specify generic service guarantee terms in *EC-Assertion* and an editor to support the automatic generation of instances of these patterns for specific SBS systems. An initial set of such patterns which specify generic security properties, including confidentiality, integrity, and availability properties, in *EC-Assertion* has been developed by Spanoudakis et al. [35]. The extension of this set is the subject of the ongoing work.

Further ongoing work on the framework focuses on its further experimental evaluation and the introduction of capabilities for probabilistic reasoning as part of the monitoring process.

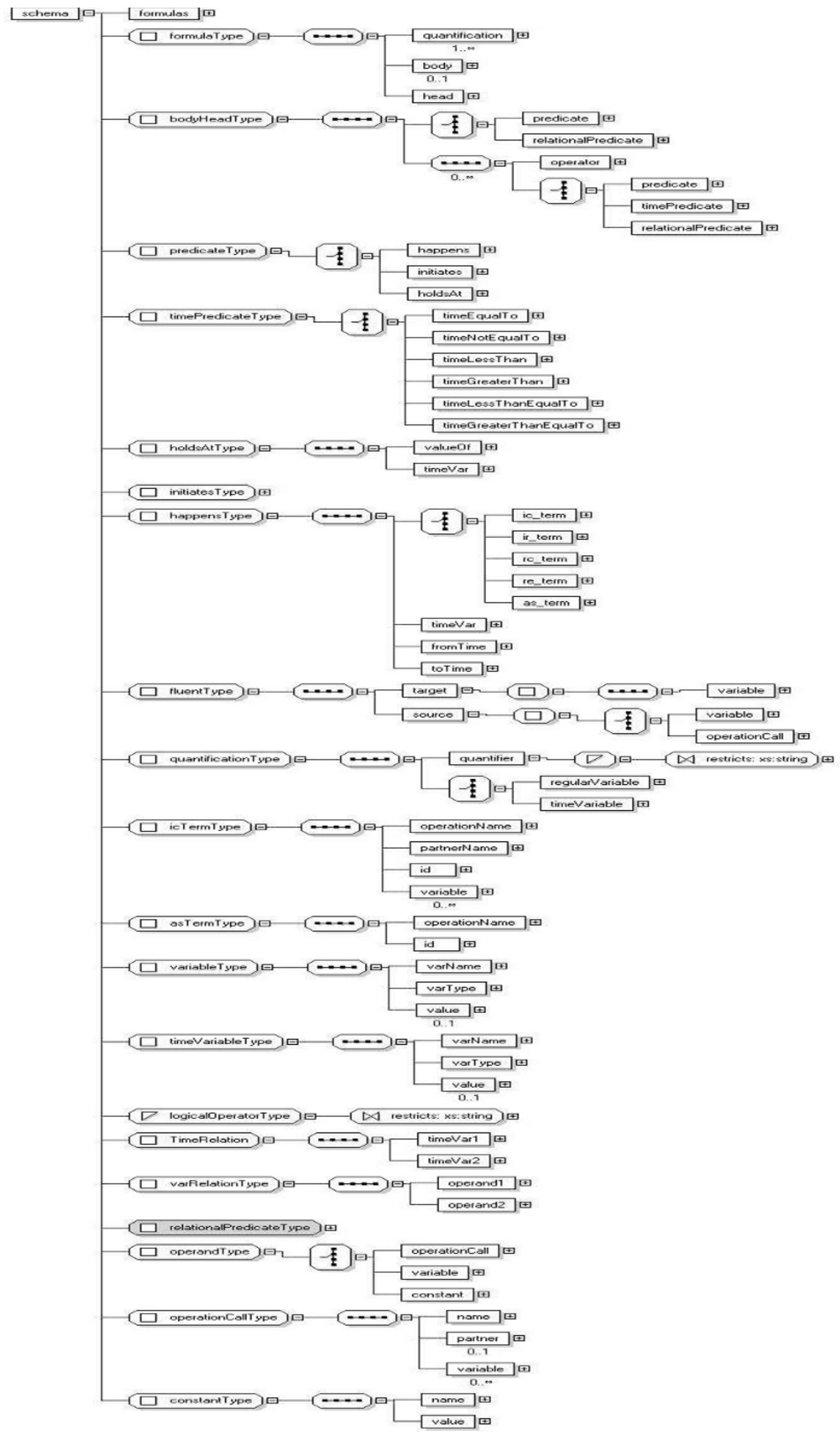
References

1. Andrews T. et al.: Business Process Execution Language for Web Services, v1.1. `http://www-106.ibm.com/developerworks/library/ws-bpel`
2. Andrieux A. et al.: Web Services Agreement Specification. Global Grid Forum, May 2004, available from: `http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf`
3. Bajaj S. et al.: Web Services Policy Framework. Sep 2004, available from: `ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf`
4. Baresi L, Guinea S, and Plebani P.: WS-Policy for Service Monitoring. 6th VLDB Workshop on Technologies for E-Services (TES-05), Trondheim, Norway, September 2–3, 2005
5. Baresi L., Ghezzi C., and Guinea S.: Smart Monitors for Composed Services. Proc. of 2nd Int. Conf. on Service Oriented Computing, New York, 2004
6. BPWS4J: `http://alphaworks.ibm.com/tech/bpws4j`
7. Brorkens M. and Moller M.: 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.
8. Brorkens M., and Moller M.: 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.

9. Capra L., Emmerich W., and Mascolo C.: Reflective middleware solutions for context-aware applications. LNCS 2192, 2001
10. Chen F. and Rosu G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. Proceedings of the 11th International Conference on Tools and Algorithms for the construction and analysis of systems, 2005
11. Clarke L. and Osterweil L.: 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.
12. CRS Case Study: Available from: `www.soi.city.ac.uk/~am697/monitoring/case_studies/CRS_Case_Study.html`
13. Dardenne A., van Lamsweerde A., and Fickas S.: Goal-Directed Requirements Acquisition. Science of Computer Programming, 20, pp. 3–50, 1993.
14. EC-Assertion: `http://www.soi.city.ac.uk/~gespan/EC-assertion.xsd`
15. Efstratiou C., Friday A., Davies N., and Cheverst K.: 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.
16. Emerging Technology Toolkit: `http://www.alphaworks.ibm.com/tech/ettk`
17. Farrell A, Sergot M., Salle M., and Bartolini C.: 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.
18. Feather M. and Fickas S.: Requirements Monitoring in Dynamic Environments. Proceedings of IEEE International Conference on Requirements Engineering, 1995
19. Feather M.S., Fickas S., Van Lamsweerde A., and Ponsard C.: Reconciling System Requirements and Runtime Behaviour. Proc. of 9th Int. Work. on Software Specification & Design, 1998.
20. Finkelstein A. and Savigni A.: A Framework for Requirements Engineering for Context-Aware Services. In Proc. of 1st Int. Workshop From Software Requirements to Architectures (STRAW 01), Toronto, Canada, May 2001.
21. Firesmith D.: Engineering Security Requirements. Journal of Object Technology, 2(1), 53–68, Jan-Feb 2003
22. Kannan S., Kim M., Lee I., Sokolsky O., and Viswanathan M.: Runtime Monitoring and Steering based on Formal Specifications. Workshop on Modeling Software System Structures in a fastly moving scenario, June 2000.
23. Keller A. and Ludwig H.: The WSLA 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.
24. Kim M., Kannan S., Lee I., Sokolsky O., and Viswanathan M.: Java-MaC: a Runtime Assurance Tool for Java Programs. In Klaus Havelund and Grigore Rosu, editors, Electronic Notes in Theoretical Computer Science, Vol. 55. Elsevier Science Publishers, 2001.
25. Lloyd J.W.: Logic for Learning: Learning Comprehensible Theories from Structured Data. Springer Verlag, ISBN 3-540-42027-4, 2003.
26. Log4j: `http://logging.apache.org/log4j/docs/`, September 2003

27. Ludwig H., Dan A., and Kearney R.: 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
28. Ludwig H., Keller A., Dan A., King R.P., and Franck R.: Web Service Level Agreement (WSLA) Language Specification, Version 1.0. IBM Corporation (January 2003), `http://www.research.ibm.com/wsla`
29. Mahbub K. and Spanoudakis G.: Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience. 3rd International IEEE Conference on Web Services (ICWS 2005), July 2005
30. OWL-S: `http://www.daml.org/services/owl-s/`
31. QTP: `http://www.soi.city.ac.uk/~am697/QTP_Case_Study.html`
32. Robinson W.: Monitoring Software Requirements using Instrumented Code. In Proc. of the Hawaii Int. Conf. on Systems Sciences, 2002.
33. Robinson W.N.: Monitoring Web Service Requirements. Proc. of 12th Int. Conf. on Requirements Engineering, 2003
34. Shanahan M.: The event calculus explained. In Artificial Intelligence Today, 409–430, Springer, 1999
35. Spanoudakis G., Kloukinas C., and Androutsopoulos K.: Towards Security Monitoring Patterns. 22nd Annual ACM Symposium on Applied Computing, Technical Track on Software Verification, March 2007
36. Spanoudakis G. and Mahbub K.: Non Intrusive Monitoring of Service Based Systems. International Journal of Cooperative Information Systems, 15(3): 325–358, 2006

A Graphical Representation of EC-Assertion


schema
formulas
formulaType
quantification
1..∞
body
0..1
head
bodyHeadType
predicate
relationalPredicate
0..∞
operator
predicate
timePredicate
relationalPredicate
predicateType
happens
initiates
holdsAt
timePredicateType
timeEqualTo
timeNotEqualTo
timeLessThan
timeGreaterThan
timeLessThanEqualTo
timeGreaterThanEqualTo
holdsAtType
valueOf
timeVar
initiatesType
happensType
ic_term
ir_term
rc_term
re_term
as_term
timeVar
fromTime
toTime
fluentType
target
variable
source
variable
operationCall
quantificationType
quantifier
restricts: xs:string
regularVariable
timeVariable
icTermType
operationName
partnerName
id
variable
0..∞
asTermType
operationName
id
variableType
varName
varType
value
0..1
timeVariableType
varName
varType
value
0..1
logicalOperatorType
restricts: xs:string
TimeRelation
timeVar1
timeVar2
varRelationType
operand1
operand2
relationalPredicateType
operandType
operationCall
variable
constant
operationCallType
name
partner
0..1
variable
0..∞
constantType
name
value