

NON INTRUSIVE MONITORING OF SERVICE BASED SYSTEMS

GEORGE SPANOUDAKIS

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

KHALED MAHBUB

*Department of Computing, City University
London, EC1V 0HB, UK
k.mahbub@soi.city.ac.uk*

Abstract. This paper presents a framework for monitoring the compliance of systems composed of web-services with requirements set for them at run-time. This framework assumes systems composed of web-services which are co-coordinated by a service composition process expressed in BPEL and uses event calculus to specify the requirements to be monitored. These requirements may include behavioural properties of a system which are automatically extracted from the specification of its composition process in BPEL and/or assumptions that system providers can specify in terms of events extracted from this specification.

1. Introduction

The verification that a software system meets its requirements at run-time has been acknowledged as a significant problem in requirements engineering research.^{1,2,3} This is because, even if it can be demonstrated that a system can meet its requirements prior to deployment, at run-time these requirements may be violated. This may be the result of unpredicted changes in the environment of a system or failure to anticipate the behaviour of all the agents interacting with it (i.e., other systems and human users¹). For software systems which are composed dynamically from *autonomous web services* co-coordinated by some *composition process* – referred to as "service based software systems" or simply *SBS systems* in the following – the ability to monitor the compliance of run-time system behaviour with requirements is even more important. This is because in such systems, both the agents interacting with the system and the individual services that constitute it that may change or behave in unpredictable ways.

The common approach that underpins techniques developed to support requirements monitoring at run-time assumes that system providers must identify the set of requirements to be monitored, specify them in some formal language, and then derive from each formal requirement statement a *pattern of events* whose occurrence at run-time would imply the violation of the requirement. The event patterns are subsequently fed into a monitor which checks if the log of events generated by the system at run-time is consistent with the requirements and reports any

[2 Draft of Paper that Appeared in International Journal of Cooperative Information Systems](#)

inconsistencies. In a monitoring setting, events are typically generated by the system to be monitored either through instrumentation (i.e., the insertion of code statements that can generate the expected events)^{1,2,3} or by querying the system if it has reflective capabilities⁴.

Existing requirements monitoring techniques fail to deal adequately with some significant complications which arise in service-based systems, as they focus on systems with no autonomous components. When, however, autonomous components exist (as in service-based systems), the failure of some of them to function as expected may lead other components to make incorrect assumptions about the state of the system and, consequently, take actions which would not have been taken if the correct state of the system was known. Furthermore, in SBS systems services may be replaced dynamically at runtime and, in cases of dynamic service binding, it might not be possible or feasible to verify that a new service satisfies the requirements set for the system and/or the service that it will replace. This may be due to a range of different reasons including, for instance, the lack of a complete specification of the new service that would enable a complete static verification of the requirements or the presence of a service specification that is too complex to allow an efficient verification within the limited time that may be available for the dynamic binding of the new service.

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

- *Car information services* (IS) which maintain registries of cars, check car availability and allocate cars to customers as requested by CRS.
- *Sensing services* (SS) which are provided by different car parks to sense cars as they are driven in or out of car parks and inform CRS accordingly.
- *User interaction services* (UI) which provide CRS with front-ends that handle interactions with the end-users.
- *Payment services* (PS) which enable CRS to collect payments for booked car rentals.

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

Covering for possibilities like this requires the introduction of types of requirements deviation beyond classical inconsistency and the development of appropriate reasoning mechanisms for detecting them.

Furthermore, the generation of events through code instrumentation may not be possible, as the provider of an SBS system might not own the individual services which constitute it and these services may change dynamically. In CRS, for example, typically the SS and IS services won't be owned by the CRS owner. In the same system, dynamic changes to the deployed services may also occur when, for example, new car rental companies and car parks make their offerings available to CRS through new IS services and services deployed by CRS may be withdrawn when companies

and car parks stop their collaboration with the system. Furthermore, the implementations of the services deployed by CRS may change even if their interface remains the same. This may for instance happen in the case of the IS services.

In such settings, monitoring has to be based on events and state information that can be obtained from the system composition process which is owned by the system provider. Requirements for individual services may still be specified and monitored in such settings but only if this is possible through events that visible to the composition process of the system, or events that can be derived from them.

In this paper, we describe a framework that supports the monitoring of *requirements* for service-based systems and provides a foundation for addressing the above limitations^a. In this framework, requirements specify *behavioural properties* of the composition process of a service-based system, or *assumptions* about the behaviour of the system as a whole, its constituent services and external agents who interact with it. Behavioural properties are automatically extracted from the specification of the composition process of the service based system which is expressed in BPEL (i.e., an XML based language for specifying executable business processes which deploy web-services to achieve their goals).⁵ Assumptions are specified by system providers in terms of events, and state variable conditions which can be checked during the operation of the system.

Both behavioural properties and assumptions are expressed in *event calculus*⁶ and runtime deviations from them are monitored by using a variant of techniques developed for checking integrity constraints in temporal deductive databases.⁷ The choice of event calculus as the requirements representation language of our framework has been motivated by the need to express the properties to be monitored in a formal language with well-defined semantics allowing: (a) the specification of temporal constraints and (b) reasoning based on the inference rules of first-order logic (this criterion has led to the choice of event calculus instead of another temporal logic language).

The framework that we have developed can detect five different types of deviations from requirements which are discussed in Section 5.

The rest of the paper is structured as follows. In Section 2, we give an overview of the monitoring framework. In Section 3, we introduce event calculus and give examples of specifying behavioural properties and assumptions using it. In Section 4, we present the scheme for extracting behavioural properties from BPEL processes and representing them in event calculus. In Section 5, we present the different types of deviations that can be detected by our framework. In Section 6, we discuss the event generation and monitoring process of the framework. In Section 7, we present an experimental evaluation of the framework and in Section 8 we overview related work. Finally, in Section 9, we discuss directions for future development of the framework.

2. Overview of Monitoring Framework

Our requirements monitoring framework has been designed with the objective to support two different monitoring scenarios for SBS systems using a *non intrusive* approach. The two key

^a An earlier version of this framework has been described in Ref. 22. This paper is an extended version of Ref. 22 that: (i) describes an amended version of the framework which supports the detection of additional types of deviations of requirements based on the use of abductive reasoning, and (ii) reports the results of an experimental evaluation of the framework.

4 [Draft of Paper that Appeared in International Journal of Cooperative Information Systems](#)

features of this approach are that monitoring is performed in parallel with the operation of an SBS system without affecting its performance and does not require the instrumentation of the composition process of an SBS system or the individual services deployed by it.

In the first of the assumed monitoring scenarios (*Scenario 1*), a human user (typically the provider of an SBS system) can request the framework to monitor whether the runtime operation of the system satisfies certain requirements and view any deviations from these requirements as soon as they are detected.

In the second scenario (*Scenario 2*), the monitoring can be requested by the environment that executes the composition process of an SBS system or a computational entity acting on its behalf for a certain period of time (or until further notice). In this scenario any deviations of the requirements which are being monitored are reported back to the system which requested the execution of the monitoring activity.

In both these scenarios, the input to the monitoring framework is a *monitoring policy*. This policy specifies:

- (i) The *BPEL process* whose execution will be monitored and the *WSDL specifications* of the web-services deployed by this process.
- (ii) The *type* of the events that can be exhibited by the environment that executes the BPEL process of the SBS system, and the *IP address* and *event port* where the stream of these events will be sent to at runtime to allow their collection by the monitoring framework.
- (iii) The *requirements* that should be monitored at runtime.
- (iv) The *reporting mode* of deviations. The reporting mode specifies the *time* between the generation of consecutive reports of deviations, and an *IP address* and *deviation port* where these deviations should be reported to.

Our monitoring framework realizes the architecture shown in Figure 1. This architecture assumes that at run-time a process execution engine executes the BPEL composition process of an SBS system and delivers its functionality^b. The framework has four main components, namely a *monitoring manager*, an *event receiver*, a *monitor*, a *graphical user interface*. It also maintains a database of the events it receives from the monitored systems and a database of the detected deviations.

^b In our current implementation, we have used the *bpws4j* engine⁸ and the Oracle BPEL process manager.

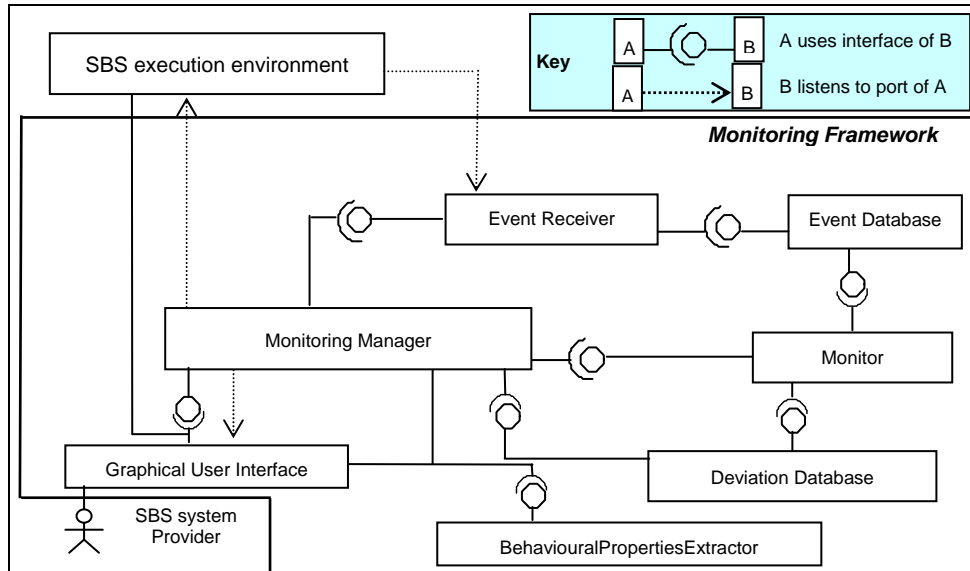


Fig.1. Monitoring framework

The *monitoring manager* is the component that has responsibility for the initiation and coordination of the monitoring process and reporting its results. Once it receives a request for starting a monitoring activity as specified by a monitoring policy, it checks whether it is possible to monitor the requirements specified in this policy given the BPEL process of the 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 requested requirements can be monitored, it starts an 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 requirements 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 other possible events that may have happened without being recorded (based on the behavioural properties and assumptions of the SBS system), 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 these requirements, 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 *behavioural properties extractor* takes as input the BPEL process of the SBS system to be monitored and generates a specification of the behavioural 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 a monitoring policy can be monitored at runtime. They are also used by the assumptions editor of the framework (see below) as primitive constructs for specifying the assumptions to be monitored in cases where the monitoring activity is initiated by a human user (see Scenario 1 above).

Finally, the framework incorporates a *graphical user interface* that gives access to the monitoring service to human users. This interface incorporates a tool that supports the specification of monitoring policies including an *assumption editor* that the user can use to specify the assumptions to be monitored and a *deviation viewer* that displays the deviations from the monitored requirements. The assumption 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 assumptions.

3. Specifications of Requirements

As we discussed in Section 1, the behavioural properties and assumptions of service-based systems that need to be monitored are expressed in event calculus (EC).

EC is a logic language based on first-order predicate calculus that can be used to represent and reason about the behaviour of dynamic systems. In EC, system behaviour is specified in terms of *events* and *fluents*.⁶ An event is something that occurs at a specific instance of time and may change the state of a system (e.g., invocation of an operation, assignment of a value to a variable) while a fluent signifies a system state. In an SBS system, such states are expressed as conditions over the values of variables of the system composition process. In CRS, for example, a fluent may express that the value of a system variable signifying whether or not a car rental request has been accepted is false.

Events and fluents, and the effects that the former have to the latter, are specified using the following EC predicates:

- **Happens**($e, t, \mathcal{R}(t_1, t_2)$) – This predicate signifies the occurrence of an event e at some time t that is within the time range $\mathcal{R}(t_1, t_2)$.^c
- **Initially** _{p} (f) – This predicate signifies that a fluent f holds at time 0.
- **Initiates**(e, f, t) – This predicate signifies that a fluent f starts to hold after the event e at time t .
- **Terminates**(e, f, t) – This predicate signifies that a fluent f ceases to hold after the occurrence of event e at time t .
- **HoldsAt**(f, t) – This predicate signifies that the fluent f holds at time t .

Fluents are specified using the following terms:

- *equalTo*(x, y) – This term signifies that the value of the fluent variable x is equal to y .
- *greaterThan*(x, y) – This term signifies that the value of the fluent variable x is greater than y .
- *lessThan*(x, y) – This term signifies that the value of the fluent variable x is less than y .

Furthermore, in the specification of behavioural properties and assumptions of SBS systems we use 5 different types of events signifying:

^c $\text{Happens}(e, t, \mathcal{R}(t_1, t_2))$ in our framework is equivalent to the formula:
 $\text{Happens}'(e, t) \wedge (t_1 \leq t) \wedge (t \leq t_2)$ where $\text{Happens}'(e, t)$ is the predicate that signifies an event occurrence in standard event calculus.

- (a) The invocation of an operation by the composition process of the system in one of its partner services. The occurrence of these events is represented by the predicate

$$Happens(ic:PartnerService:Operation(_old, _inVar), t, \mathfrak{H}(t, t))$$

The term $ic:PartnerService:Operation(_old, _inVar)$ in this predicate represents the invocation event. In this term, $Operation$ is the name of the invoked operation, $PartnerService$ is the name of the service that provides $Operation$, $_old$ is a variable whose value determines the exact instance of the invocation of $Operation$ within a specific instance of the execution of the composition process of the SBS system, and $_inVar$ is a variable whose value is the value of the input parameter of $Operation$ at the time of its invocation.

- (b) The return from the execution of an operation invoked by the composition process in a partner service. The occurrence of these events is represented by the predicate:

$$Happens(ir:PartnerService:Operation(_old), t, \mathfrak{H}(t, t))$$

The term $ir:PartnerService:Operation(_old)$ in this predicate represents the return event. $PartnerService$, $Operation$ and $_old$ in this term are as defined in (a). In cases where $Operation$ has an output variable $_outVar$, the value of this variable at the return of the operation is represented by the predicate

$$Initiates(ir:PartnerService:Operation(_old), equalTo(outVar, _outVar), t)$$

This predicate expresses the initialization of a fluent variable ($outVar$) with the value of $_outVar$.

- (c) The invocation of an operation in the composition process by a partner service. The occurrence of these events is represented by the predicate:

$$Happens(rc:PartnerService:Operation(_old), t, \mathfrak{H}(t, t))$$

The term $rc:PartnerService:Operation(_old)$ in this predicate represents the invocation event. $Operation$ and $_old$ are as defined in (a) and $PartnerService$ is the name of the service that invokes the operation. In cases where $Operation$ has an input variable $_inVar$, the value of this variable at the time of its invocation is represented by the predicate

$$Initiates(rc:PartnerService:Operation(_old), equalTo(inVar, _inVar), t)$$

This predicate expresses the initialization of a fluent variable $inVar$ with the value of $_inVar$.

- (d) The reply following the execution of an operation that was invoked by a partner service in the composition process. The occurrence of these events is represented by the predicate:

$$Happens(re:PartnerService:Operation(_old, _outVar), t, \mathfrak{H}(t, t))$$

The term $re:PartnerService:Operation(_old, _outVar)$ in this predicate represents the reply event. In this term, $Operation$ and $_old$ are as defined in (a), $PartnerService$ is the name of the service that invoked $Operation$, and $_outVar$ is a variable whose value is the value of the output parameter of the operation at the time of the reply.

- (e) The assignment of a value to a variable. The occurrence of these events is represented by the predicate:

$$Happens(as:AssignmentName(_aId), t, \mathfrak{H}(t, t))$$

The term $as:AssignmentName(_aId)$ in this predicate represents the assignment event. In this term, $AssignmentName$ is the name of the assignment in the composition process of an SBS

system and *_ald* is a variable whose value identifies the exact instance of the assignment within a specific instance of the execution of this process.

The restriction of the events that may be used in the specification of behavioural properties and assumptions to the above types guarantees that the specified properties will be monitorable without the need to instrument the services deployed by an SBS system.

In addition to the above predicates and event/fluent denoting terms, EC formulas can use the predicates $<$, $=$ 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 $t2$). Examples of specifications of behavioural properties and assumptions of service-based systems in event calculus are shown in Figure 2. All the behavioural properties and assumptions shown in this figure are related to the car rental system (CRS) introduced in Section 1.

The formula *B2*, for example, expresses a behavioural property of the composition process of CRS. *B2* specifies that when CRS replies to a customer rental request (see the predicate $\text{Happens}(rc:UI:CarRequest(_oID1, _vID, _pID), t1, \mathfrak{R}(t1, t1))$) to confirm the rental of the car *_vID* from a car park *_pID*, it waits for the invocation of the operation *Depart* in its composition process to get confirmation that the rented car has departed from the relevant car park for 6 time units and, if this happens, it invokes the operation *MakeUnavailable* in the IS service to record the relevant car as unavailable. The invocation of the operation *Depart* is expressed by the predicate $\text{Happens}(rc:SS:Depart(_oID2), t2, \mathfrak{R}(t1, t1+6*t_u))$ in the formula. This operation has two input parameters whose values must be equal to the values of the variables *_vID* and *_pID* in the formula to ensure that the call of *Depart* signifies the departure of the car whose rental was confirmed by the call of *CarRequest*. Furthermore, the values of these variables are assigned to the fluent variables *v* and *p* which keep a record of the last car and car park reported by a call of the *Depart* operation, respectively. These assignments are expressed by the predicates $\text{Initiates}(rc:SS:Depart(_oID2), \text{equalTo}(v, _vID), t3)$ and $\text{Initiates}(rc:SS:Depart(_oID2), \text{equalTo}(p, _pID), t3)$ in the formula. Finally, the invocation of the operation *MakeUnavailable* is expressed by the predicate $\text{Happens}(ic:IS:MakeUnavailable(_oID3, _vID, _pID), t3, \mathfrak{R}(t1+7*t_u, t1+7*t_u))$ of *B2*.

The formula *A1* in Figure 2 expresses an assumption about the behaviour of the sensing services (SS) of CRS. According to this formula, if a car *_vID* is sensed to enter a car park *_pID1* at some time $t1$ and later at some time $t2$ the same car is sensed to enter in the same or a different car park, then a *Depart* event must have also occurred between the two enter events to signify the departure of *_vID* from *_pID1*. The Happens predicates in *A1* represent the invocation of the operations *Enter* and *Depart* in CRS by SS following the entrance and departure of cars in car parks. The predicate $\text{Happens}(rc:SS:Enter(_oID1), t1, \mathfrak{R}(t1, t1))$, for example, represents the invocation of the operation *Enter* in CRS following the entrance of a car in a car park. The Initiates predicates that have a specific operation invocation event as arguments represent the binding of the parameters of this operation to specific values. Thus, the predicates $\text{Initiates}(rc:SS:Enter(_oID1), \text{equalTo}(v, _vID), t1)$ and $\text{Initiates}(rc:SS:Enter(_oID1), \text{equalTo}(p, _pID1), t1)$ initiate fluents representing the binding of the parameters *v* and *p* of the operation *Enter* when this operation is invoked.

Some additional restrictions apply to the specification of formulas in our framework. More specifically, if the time variable t of a Happens predicate is existentially quantified, at least one of the lower boundary (LB) or upper boundary (UB) of its time range $\mathfrak{R}(LB, UB)$ must be specified.

This is possible by using: (i) constant time values or (ii) arithmetic expressions of time variables t' of other *Happens* predicates of the same formula (in this case t will said to be constrained by t'). If t is a universally quantified variable then any of the boundaries LB and UB may be unspecified. In this case the time variable is unconstrained and the *Happens* predicate has the form $\text{Happens}(e, t, \mathfrak{R}(t, t))$. Furthermore, a formula is valid if the time variables of all 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 the EC formulas. A specification in our framework must also be compliant with the standard axiomatic definition of EC shown in Figure 3.



Fig.2. Behavioural properties and assumptions of CRS

(EC1) $\text{Clipped}(t1, f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Terminates}(e, f, t)$
(EC2) $\text{Declipped}(t1, f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Initiates}(e, f, t)$
(EC3) $\text{HoldsAt}(f, t) \Leftarrow \text{Initially}_p(f) \wedge \neg \text{Clipped}(0, f, t)$
(EC4) $\text{HoldsAt}(f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Initiates}(e, f, t) \wedge \neg \text{Clipped}(t, f, t2)$
(EC5) $\neg \text{HoldsAt}(f, t2) \Leftarrow (\exists e, t) \text{Happens}(e, t, \mathfrak{R}(t1, t2)) \wedge \text{Terminates}(e, f, t) \wedge \neg \text{Declipped}(t, f, t2)$
(EC6) $\neg \text{HoldsAt}(f, t) \Leftarrow \text{Initially}_n(f) \wedge \neg \text{Declipped}(0, f, t)$
(EC7) $\text{HoldsAt}(f, t2) \Leftarrow \text{HoldsAt}(f, t1) \wedge (t1 < t2) \wedge \neg \text{Clipped}(t1, f, t2)$
(EC8) $\neg \text{HoldsAt}(f, t2) \Leftarrow \neg \text{HoldsAt}(f, t1) \wedge (t1 < t2) \wedge \neg \text{Declipped}(t1, f, t2)$
(EC9) $\text{Happens}(e, t, \mathfrak{R}(t1, t2)) \Rightarrow (t1 \leq t2) \wedge (t1 \leq t) \wedge (t \leq t2)$

Fig.3. Axioms of event calculus

4. Extraction of Requirements from BPEL

As discussed in Section 2, our framework extracts the behavioural properties of SBS systems from their BPEL process. Following the extraction of such properties, assumptions are specified by system providers in terms of event and fluent predicates that have been extracted from the BPEL specification. The extraction of behavioural properties from this specification is discussed in the following.

The specification of an SBS composition process in BPEL defines: (a) the *partners* of the process (i.e. the web services that invoke the process or are invoked by it); and (b) *variables* storing the contents of messages which are exchanged between the process and its constituent web-services. Variables can be accessed at different stages in the execution of the composition process in order to direct appropriately the subsequent flow of control.

A BPEL SBS system composition process is specified using two kinds of activities, namely *basic* and *structured* activities.

Basic activities in BPEL express primitive functions such as the invocation of operations and assignments of variable values. More specifically, a basic activity can be:

- an *invoke* activity – this activity is used to call an operation in one of the partner services of the SBS composition process;
- a *receive* activity – this activity makes the SBS composition process to wait for the receipt of an invocation of one of its operations by some of its partner services;
- a *reply* activity – this activity allows the SBS composition process to respond to a request for the execution of an operation previously accepted through a *receive* activity;
- an *assign* activity – this activity is used to copy the value of one SBS composition process variable to another variable;
- a *throw* activity – this activity is used to signal an internal fault; or
- a *wait* activity – this activity forces the composition process to remain idle for a certain period of time.

Basic BPEL activities are transformed into EC formulas according to the transformations shown in Figure 4.

As shown in this figure, an *invoke* activity that calls an operation O in a partner service P is represented in EC as a conjunction of a predicate that signifies the event of calling O at some time $t1$ (i.e., $\text{Happens}(ic:P:O(_ID, _X), t1, \mathfrak{R}(t1, t1))$), a predicate that signifies the event of the

notification of the completion of O to the composition process at some time $t2$ after $t1$ (i.e., $\text{Happens}(ir:P:O(_ID), t2, \mathcal{R}(t2, t2))$), and a predicate that signifies the initiation of a fluent representing the value of the output variable $_Y$ of O upon its return (i.e., the predicate $\text{Initiates}(ir:P:O(_ID), \text{equalTo}(Y, _Y), t2)$). It should be noted that in the EC formula for *invoke*, the variable $_ID$ takes as value a unique identifier that represents the exact instance of the operation invocation in the SBS composition process, and the variable $_X$ takes the value that the input variable X of O has at the time of the invocation. Note that in this case, it is unrealistic to predict the precise time of the completion of the execution and return of O (due to the physical distribution of component services and/or network communication delays). Thus, the EC formula that is automatically extracted for *invoke* does not specify an upper limit for the time variable $t2$ which signifies the time of the return of O (the range of $t2$ is specified as $\mathcal{R}(t1, t2)$ in the relevant formula of Figure 4).

BPEL Basic Activity	EC Representation
<code><invoke partner="P" portType="a:Pport" operation="O" inputVariable="X" outputVariable="Y"/></code>	Happens (ic:P:O($_ID$, $_X$), $t1, \mathcal{R}(t1, t1)$) \wedge ($\exists t2$) Happens (ir:P:O($_ID$), $t2, \mathcal{R}(t1, t2)$) \wedge Initiates (ir:P:O($_ID$), $\text{equalTo}(Y, _Y), t2$)
<code><reply partner="P" portType="a:Pport" operation="O" variable="X"/></code>	Happens (re:P:O($_ID$, $_X$), $t, \mathcal{R}(t, t)$)
<code><receive partner="P" portType="a:Pport" operation="O" variable="X" ... /></code>	Happens (rc:P:O($_ID$), $t, \mathcal{R}(t, t)$) \wedge Initiates (rc:P:O($_ID$), $\text{equalTo}(X, _X), t$)
<code><assign name="A"> <copy><from variable="X" part="a"/> <to variable="Y" part="b"/> </copy></assign></code>	Happens (as:A($_ID$), $t1, \mathcal{R}(t1, t1)$) \wedge HoldsAt ($\text{equalTo}(x.a, _X.a)$, $t1$) \wedge ($\exists t2$) ($t1 < t2$) \wedge Initiates (as:A($_ID$), $\text{equalTo}(Y.b, _X.a), t2$)
<code><throw faultName="fN" faultVariable="X"/></code>	Happens (th:fN($_ID$, $_X$), $t, \mathcal{R}(t, t)$) or Happens (th:fN($_ID$), $t, \mathcal{R}(t, t)$) if no fault variable X is specified.
<code><actType name="A">...</actType> <wait for="T"/> <actType name="B">...</actType></code>	EC (A, []) \wedge EC (B, []) \wedge $\text{max}(A) + T < \text{min}(B)$

Fig. 4. Transformations of basic BPEL activities to EC formulas

Similarly, a *receive* activity is represented by a predicate signifying the receipt of the invocation of an operation O of the SBS composition process by a partner service P (see the predicate $\text{Happens}(rc:P:O(_ID), t, \mathcal{R}(t, t))$ in Figure 4), and a predicate that initiates a fluent representing the value of the input variable of O at the time of the call (see predicate $\text{Initiates}(rc:P:O(_ID), \text{equalTo}(X, _X), t)$ in Figure 4).

The use of the *Initiates* predicate in both the case of *invoke* and *receive* activities is based on the principle that value bindings of variables which are visible to the BPEL process should be represented by fluents in order to be accessible to the reasoning process that checks the satisfiability of formulas.

The same principle underpins the representation of *assignment* activities in EC which, as shown in Figure 4, is a conjunction of a predicate that signifies the assignment event (i.e., $\text{Happens}(as:A(_ID), t1, \mathcal{R}(t1, t1))$), a predicate that signifies the value of the source variable of the assignment (i.e., $\text{HoldsAt}(\text{equalTo}(x.a, _X.a), t1)$) and a predicate that signifies the

assignment of this value to the target variable of the assignment (i.e., $\text{Initiates}(\text{as}:\text{A}(_ID), \text{equalTo}(\text{Y.b}, _X.a), t_2)$).

A *reply* activity responding to the invocation of an operation O in the SBS composition process is represented by a `Happens` predicate signifying the occurrence of an event which notifies the completion of the execution of O and returns its results as the value of the output variable X of O .

BPEL Structured Activity	EC Representation
<pre><sequence> <actType name="A">...</actType> <actType name="B">...</actType> ... </sequence></pre>	$\text{EC}(\text{A}, []) \Rightarrow \text{EC}(\text{B}, [^*]) \wedge \max_t(\text{A}) < \min_t(\text{B})$
<pre><flow> <links> <link name="AtoB"/> <link name="AtoC"/> ... </links> <actType name="A"> <source linkName="AtoB" transitionCondition="P=v1"/> <source linkName="AtoC" /> ... </actType> <actType name="B"> <target linkName="AtoB" /> ... </actType> <actType name="C"> <target linkName="AtoC" /> ... </actType> <actType name="D">... </actType> </flow></pre>	$\text{EC}(\text{A}, []) \wedge$ $\text{HoldsAt}(\text{equalTo}(\text{P}, _v_1), t_1) \wedge \max_t(\text{A}) < t_2$ $\Rightarrow \text{EC}(\text{B}, [\min_t(\text{B})]) \wedge t_2 < \min_t(\text{B})$ $\text{EC}(\text{A}, []) \Rightarrow$ $\text{EC}(\text{C}, [\min_t(\text{C})]) \wedge$ $\max_t(\text{A}) < \min_t(\text{C})$ $\text{EC}(\text{D})$
<pre><while condition="P=v1"> <actType name="A"> ... </actType> </while></pre>	$\text{HoldsAt}(\text{equalTo}(\text{P}, v_1), t) \Rightarrow \text{EC}(\text{A}, [\min_t(\text{A})]) \wedge t < \min_t(\text{A})$
<pre><sequence> <actType name="A"> ... </actType> <pick> <onMessage partner="P" portType="a:Pport" operation="O" variable="X"> <actType name="B"> ... </actType> </onMessage> <onAlarm for="T"> <actType name="C"> ... </actType> </onAlarm> </pick> </sequence></pre>	$\text{EC}(\text{A}, []) \wedge \text{Happens}(\text{om}:\text{O}(_vID, vX), t_2, \mathfrak{R}(\max_t(\text{A}), \max_t(\text{A}) + T)) \wedge \text{Initiates}(\text{om}:\text{O}(_vID, vX), \text{equalTo}(\text{X}, _X), t_2) \Rightarrow$ $\text{EC}(\text{B}, [\min_t(\text{B})]) \wedge t_1 < \min_t(\text{B})$ $\text{EC}(\text{A}, []) \wedge \neg \text{Happens}(\text{om}:\text{O}(_vID, vX), t_2, \mathfrak{R}(\max_t(\text{A}), \max_t(\text{A}) + T)) \Rightarrow \text{EC}(\text{C}, [\min_t(\text{C})]) \wedge \max_t(\text{A}) + T < \min_t(\text{C})$

Fig. 5. Patterns for transforming structured BPEL structured activities to EC

Throw activities are represented by the predicate $\text{Happens}(\text{th}:\text{fN}(_ID, _X), t, \mathfrak{R}(t, t))$. The term $\text{th}:\text{fN}(_ID, _X)$ in this predicate signifies the generation of a fault signal (this is indicated by the type th of the term) whose name is fN at the time point t . The variable $_ID$ in this term takes as value the unique identifier that is generated for the specific fault during the execution of the BPEL process and the variable $_X$ takes as value the data that are attached to the fault to allow fault handlers deal with it. In cases where a *throw* activity does not specify any such data, the term representing the fault is simplified to $\text{th}:\text{fN}(_ID)$.

Finally, *wait* activities are represented by a time constraint requiring that the value of the time variable of the latest predicate in the EC formula representing the activity before it (i.e., $\max_t(\text{A})$) should be less or equal to the value of the time variable of the earliest predicate in the formula

representing the activity after the wait (i.e., $min_t(B)$) (the terms $EC(A, [I])$ and $EC(B, [I])$ in the pattern for a wait activity are explained below).

Structured activities in BPEL provide the control and data flow structures that enable the coordination of basic activities into an SBS composition process. A structured activity in BPEL may be:

- A *sequence* activity – this activity specifies an ordered list of other activities that must be performed sequentially.
- A *switch* activity – this activity specifies an ordered list of one or more conditional branches that include other activities and may be executed subject to the satisfiability of the conditions associated with them.
- A *flow* activity – this activity specifies a set of two or more other activities that should be executed concurrently. A *flow* activity completes when all of the activities in it have completed. Synchronization dependencies between activities inside a flow can be specified using *links*. Each link defines a *target* activity that cannot start before the completion of a *source* activity which is also defined by the link.
- A *pick* activity – this activity forces the composition process to wait for different events and perform different activities associated with each of these events as soon as it occurs.
- A *while* activity – this activity is used to specify the iterative execution of one or more activities for as long as a condition is true.

Figure 5 presents examples of transformation patterns which are applied to transform BPEL *sequence*, *flow*, *while* and *pick* activities into EC formulas. In these patterns: (i) *actType* can be a basic or structured BPEL activity of any type; (ii) $EC(A, [t_1, \dots, t_n])$ represents the EC formulas that an activity *A* is transformed to after replacing the quantifiers of all universally quantified time variables in $[t_1, \dots, t_n]$ with the existential quantifier^d; (iii) $min_t(X)$ represents the time variable of the earliest predicate in the formulas of activity *X* (i.e., the predicate that is expected to occur first given the constraints of the time variables of the predicates representing *X*), and (iv) $max_t(X)$ represents the time variable of the latest predicate in the formulas representing *X* (i.e., the predicate that is expected to occur last given the constraints of the time variables of the predicates representing *X*).

An example of EC formulas extracted according to the patterns in Figures 4 and 5 is shown in Figure 6 below. This figure shows an extract of the BPEL specification of the composition process of the CRS system that we introduced in Section 1, and the EC formula extracted from it. The extract refers to the part of the process that receives a request for a car and checks for available cars.

Part of BPEL composition process for CRS	
<pre> <process name="CRS" .../> <partners> ... </partners> ... <flow> <links> <link name="receive-to-auth"/> ... </links> <receive name="receiveRequest" partner="UI" portType="sns:CRSUI" operation="CarRequest" variable="Req" createInstance="yes"> <source linkName="receive-to-auth"/> </correlations> ... </correlations> </pre>	

^d $EC(A, [I])$ indicates that there should be no changes to the quantifiers of universally quantified time variables in *A* and $EC(A, [I^*])$ indicates that all the universally quantified time variables in *A* should be existentially quantified in the formula resulting from the transformation.

<pre> </receive> <sequence> <target linkName="receive-to-auth" /> <assign name="a1"> <copy> <from variable="Req" part="Loc" /> <to variable="Q" part="Loc" /> </copy> </assign> <invoke name="findCar" partner="IS" portType="crns:CRSIS" operation="FindAvailable" inputVariable="Q" outputVariable="Res"> ... </invoke> </sequence> ... </flow> ...</process> </pre>
<p>EC formulas</p> <p>Happens(rc:Ul:CarRequest(_oID1),t1,R(t1,t1)) \wedge Initiates(rc:Ul:CarRequest(_oID1),equalTo(Req.Loc, _Req.Loc), t1) \wedge Initiates(rc:Ul:CarRequest(_oID1),equalTo(Req.CId, _Req.CId),t1) $\Rightarrow ((\exists t2)$</p> <p>Happens(as:a1(_aID),t2,$\mathfrak{R}(t1,t2)$) \wedge HoldsAt(equalTo(Req.loc,_loc),t2) \wedge</p> <p>($\exists t3$) (t2 < t3) Initiates(as:a1(_aID),equalTo(Q.Loc, _loc),t3) \wedge</p> <p>($\exists t4$) Happens(ir:IS:FindAvailable(_oID2, _Q),t4,$\mathfrak{R}(t3,t4)$) \wedge</p> <p>($\exists t5$) Happens(ir:IS:FindAvailable(_oID2),t5, $\mathfrak{R}(t4,t5)$) \wedge</p> <p>Initiates(ir:IS:FindAvailable(_oID2), equalTo(Res,_Res), t5))</p>

Fig. 6. Example of EC formulas extracted from the BPEL process for CRS

The first implication (\Rightarrow) in the extracted EC formula of Figure 6 represents the link *receive-to-auth* in the flow activity of the process. The conditions of this implication represent the receive activity *receiveRequest*, and its consequence represents the sequence activity in the process. The first three conditions in the second implication represent the assign activity *a1* and the latter three predicates represent the invoke activity *findCar*.

Note that in the EC formula, due to the patterns introduced in Figures 4 and 5, all time variables but *t1* which is the variable of the predicates representing the activity *receiveRequest*, which is the first to be executed in the process, are existentially quantified, and constrained to take values greater than *t1*. These constraints are necessary in order to reflect the ordering of activities in the BPEL process. Note also that the variable *t1* is universally quantified in order to allow for cases where the process never receives a request to execute the operation *CarRequest*. Finally, the existential quantification of the other variables in the formula reflects the fact that all the subsequent activities must be executed following the receipt of the car request.

5. Deviations

Our monitoring framework can detect five types of deviations of the behaviour of an SBS system from the behavioural properties and assumptions set for it. These types are: (i) inconsistencies evidenced from the recorded system behaviour, (ii) inconsistencies evidenced from the expected system behaviour, (iii) cases of unjustified system behaviour, (iv) possible inconsistencies evidenced from the expected system behaviour, and (v) possible cases of unjustified system behaviour.

The deviations of the first of these types are detected by using only the events which are generated by an SBS system at run-time (i.e., the recorded behaviour of a system). The deviations of the remaining four types are detected by using additional events which are derived from the recorded behaviour of a system by deductive reasoning (in the case of types (ii) and (iii)) or a combination of deductive and abductive reasoning (in the case of types (iv) and (v)). In the rest of this section, we define these types of deviations and give examples of them.

5.1. Inconsistencies Evidenced from the Recorded Behaviour of a System

The recorded behaviour of a system is defined as follows:

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

On the basis of the above definition of recorded behaviour, inconsistencies evidenced from this behaviour are defined as follows:

Definition 2: An assumption f of the form $f: H \Rightarrow B$ is *inconsistent* with the recorded behaviour of a system S until time T if and only if: $\{E_R(T), EC_a\} \models \neg f$ where

- (i) \models signifies logical entailment using also the principle of negation as failure, and
- (ii) EC_a are the axioms of event calculus. ■

Example. Suppose that the log of events of the CRS system includes the literals shown in Figure 7. Given these events, the recorded behaviour of CRS is inconsistent with assumption A2. According to this assumption, within 10 time units following the entrance of a car $_vID$ in a car park $_pID$, signified by the invocation of the operation *Enter* in the composition process of CRS, the operation *RetKey* should be invoked in the same process to signify the return of the key of the relevant car. A violation of A2 can be detected at time $t=38$. At this time point, we can deduce by negation as failure the predicate $\neg \text{Happens}(rc:UI:RetKey(ID), t, \mathcal{R}(27, 37))$, which in conjunction with the literals L11, L12 and L13 in E_R entails $\neg A2$. In this case, the inconsistency has been caused by the failure of a CRS customer to return the key of car that he/she has driven back to a car park operated by CRS within 10 time units from the time that the car entered the car park as expected by A2.

5.2. Inconsistencies Evidenced from the Expected Behaviour of a System

The second type of deviations that can be detected in our framework are inconsistencies between the behavioural properties (B_S) or assumptions (A_S) of an SBS system S and its recorded and expected behaviour. The "expected behaviour" of an SBS system includes the set of predicates that can be derived by deductive reasoning from its recorded behaviour using the formulas in B_S and A_S . Thus, the expected behaviour is the behaviour that should have been exhibited by an SBS system had all the behavioural properties and assumptions set for it been realized. The inconsistencies evidenced from the expected behaviour of an SBS system are formally defined as follows:

Definition 3: Given a set of behavioural properties B_S and a set of assumptions A_S of an SBS system S , a behavioural property or assumption of S of the form $f: C \Rightarrow A$ is inconsistent with the *expected behaviour* of S at time T if and only if:

$$\{E_R(T), EC_a, \text{dep}((B_S \cup A_S) - \{f\}, f)\} \models \neg f \quad \text{---}$$

where $\text{dep}((B_S \cup A_S) - \{f\}, f)$ is the set of formulas $g: B \Rightarrow H$ in $(B_S \cup A_S) - \{f\}$ which f depends on. 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 ■

L1	:	Happens (rc:UI:RetKey(ID6),15, $\mathfrak{R}(15,15)$)
L2	:	Initiates (rc:UI:RetKey(ID6), equalTo(v, v1), 15)
L3	:	Initiates (rc:UI:RetKey(ID6), equalTo(p, p1), 15)
L4	:	Happens (rc:SS:Enter(ID7),18, $\mathfrak{R}(18,18)$)
L5	:	Initiates (rc:SS:Enter(ID7), equalTo(v, v2), 18)
L6	:	Initiates (rc:SS:Enter(ID7), equalTo(p, p2), 18)
L7	:	Happens (rc:UI:RetKey(ID8),23, $\mathfrak{R}(23,23)$)
L8	:	Initiates (rc:UI:RetKey(ID8), equalTo(v,v2), 23)
L9	:	Initiates (rc:UI:RetKey(ID8), equalTo(p, p2), 23)
L10	:	Happens (ic:IS:Available(ID9,v2,l2),26, $\mathfrak{R}(26,26)$)
L11	:	Happens (rc:SS:Enter(ID10),27, $\mathfrak{R}(27,27)$)
L12	:	Initiates (rc:SS:Enter(ID10), equalTo(v, v1),27)
L13	:	Initiates (rc:SS:Enter(ID10), equalTo(p, p1),27)
L14	:	Happens (rc:UI:CarRequest(ID11),28, $\mathfrak{R}(28,28)$)
L15	:	Initiates (rc:UI:CarRequest(ID11), equalTo(c,c2),28)
L16	:	Initiates (rc:UI:CarRequest(ID11), equalTo(p,p2),28)
L17	:	Happens (ic:IS:FindAvailable(ID12,p2),29, $\mathfrak{R}(29,29)$)
L18	:	Happens (ir:IS:FindAvailable(ID12),30, $\mathfrak{R}(30,30)$)
L19	:	Initiates (ir:IS:FindAvailable(ID12), equalTo(v,v2),30)
L20	:	Happens (re:UI:CarRequest(ID13,v2,p2),31, $\mathfrak{R}(31,31)$)
L21	:	Happens (ic:IS:Available(ID14,v2,p2),38, $\mathfrak{R}(38,38)$)
L22	:	Happens (rc:UI:CarRequest(ID15),40, $\mathfrak{R}(40,40)$)
L23	:	Initiates (rc:UI:CarRequest(ID15), equalTo(c,c3),40)
L24	:	Initiates (rc:UI:CarRequest(ID15), equalTo(p,p2),40)
L25	:	Happens (ic:IS:FindAvailable(ID16,p2),41, $\mathfrak{R}(41,41)$)
L26	:	Happens (ir:IS:FindAvailable(ID16),42, $\mathfrak{R}(42,42)$)
L27	:	Initiates (ir:IS:FindAvailable(ID16), equalTo(v,v2),42)
L28	:	Happens (rc:UI:RetKey(ID17),43, $\mathfrak{R}(43,43)$)
L29	:	Initiates (rc:UI:RetKey(ID17),equalTo(v,v2),43)
L30	:	Initiates (rc:UI:RetKey(ID17),equalTo(p,p2),43)
L31	:	Happens (re:UI:CarRequest(ID18,v2,p2),44, $\mathfrak{R}(44,44)$)

Fig.7. Event log of a car rental system

According to this definition, the check about the inconsistency of a formula f with the expected behaviour of a system must, in addition to events which are recorded in $E_R(T)$, take into account events which can be derived from other formulas in $B_S \cup A_S$ and can affect the satisfiability of f . The definition of the set $dep((B_S \cup A_S) - \{f\}, f)$ in Definition 3 is similar to the notion of *direct* and *indirect dependence* in Ref. 7.

Example. Given the event log of Figure 7, the assumption $A3$ is violated by the expected behaviour of CRS at $t=43$. $A3$ is an assumption about the behaviour of the IS service stating that when IS executes the operation *FindAvailable* it should not report a car as available unless this is indeed the case. This violation arises as the negation of the assumption $A3$, i.e.,

$$(\forall t1:\text{Time}) \text{Happens}(\text{ir:IS:FindAvailable}(_oID1), t1, \mathfrak{R}(t1, t1)) \wedge \\ \text{HoldsAt}(\text{equalTo}(\text{Availability}(_vID), \text{"not avail"}), t1 - t_u) \wedge \\ \text{Initiates}(\text{ir:IS:FindAvailable}(_oID1), \text{equalTo}(v, _vID), t1)$$

is entailed by the events in $E_R(44)$ and predicates that can be deduced from other assumptions that $A3$ depends on and the axioms of event calculus.

More specifically, the negation of $A3$ can be derived from:

- (i) the literal L26 in $E_R(44)$
- (ii) the literal L27 in $E_R(44)$; and
- (iii) the predicate $\text{HoldsAt}(\text{equalTo}(\text{availability}(v2), \text{"not avail"}), 41)$.

The latter predicate is derived as follows:

- (iv) from the literal L20 and assumption $A5$ we can derive
 - (DL1) $\text{Happens}(\text{rc:SS:Depart}(\text{doID1}), t2, \mathfrak{R}(32, 41))$
 - (DL2) $\text{Initiates}(\text{rc:SS:Depart}(\text{doID1}), \text{equalTo}(v, v2), t2)$

- (DL3) $\text{Initiates}(\text{rc:SS:Depart}(\text{doID1}), \text{equalTo}(\text{p}, \text{p2}), \text{t2})$
 (v) from (DL1)–(DL2), the literals L28 and L29, and assumption A4 we can derive
 (DL4) $\text{HoldsAt}(\text{equalTo}(\text{availability}(\text{v2}), \text{"not avail"}), \text{t}) \wedge (41 \leq \text{t}) \wedge (\text{t} \leq 43)$

In this example, the inconsistency is caused by the failure of the SS service to send an $\text{rc:SS:Depart}(\text{ID})$ event to CRS following the event $\text{Happens}(\text{re:UI:CarRequest}(\text{ID13}, \text{v2}, \text{p2}), 31, \mathcal{R}(31, 31))$ as expected by A5. Thus, according to B1, CRS invoked the operation *Available* to mark the vehicle v2 as available (see the literal L21 in Figure 7). Then, when the operation $\text{ic:IS:FindAvailable}(\text{ID16}, \text{p2})$ was invoked in IS at $t=42$ (see literal L26 in Figure 7), IS reported v2 as an available vehicle. However, according to A4, v2 should not be available between its (non-reported) departure and the return of its key at $t=43$.

5.3. Unjustified Behaviour

The third type of deviations that can be detected by our framework occurs when the conditions of a behavioural property f that has generated an event e are satisfied by the recorded system behaviour but violated by the expected system behaviour. In such cases, the generation of the event e is the result of an assumption that an SBS system makes about the satisfiability of the conditions of f at run-time which is not, however, consistent with assumptions given for the services that it deploys. Such cases constitute what we refer to as "unjustified behaviour". This type of deviation is formally defined as follows:

Definition 4: Given a set of behavioural properties B_S and a set of assumptions A_S of an SBS system S , a behavioural property of S of the form $f:B \Rightarrow H$ is said to generate *unjustified behaviour* if and only if there is a literal e such that

- (i) $e \in E_R(T)$
- (ii) e can be unified with H
- (iii) $\{E_R(T) - \{e\}, B_S, EC_a\} \models e$
- (iv) $\{E_R(T) - \{e\}, B_S - \{f\}, EC_a\} \not\models e$ and
- (v) there is a predicate L in B for which, $\{E_R(T), \text{dep}((B_S \cup A_S) - \{f\}, f), EC_a\} \models \neg L$ ■

The conditions (i)-(iv) in Definition 4 identify an event e which has been generated by the system due to the realisation of a formula f (condition (iv) guarantees that e cannot have been generated by some other formula). The satisfaction of conditions (i)-(iv) implies that the conditions of f are satisfied by the recorded behaviour of the system. Note, however, that according to condition (v), there is some condition in B that would not be satisfied if all the events that could be generated by formulas which f depends on are taken into account. In such cases, e is the result of behaviour that is based on wrong assumptions about the satisfiability of the conditions of f that the system made at run-time.

Example. At $t=44$, the reply produced by CRS to a car rental request that had been received earlier and is represented by the literal L31 can be detected as a case of unjustified behaviour. This reply was generated due to B3 and reported that car v2 as available for hire at car park p2.

The case of unjustified behaviour in this instance is caused because the event $\text{Happens}(\text{re:UI:CarRequest}(\text{ID18}, \text{v2}, \text{p2}), 44, \mathcal{R}(44, 44))$ could only have been generated by B3 (no other formula in the set of behavioural properties of Figure 2 has a `Happens` predicate

signifying the occurrence of `re:UI:CarRequest` events in its head), and one of the conditions of $B3$ is violated by the expected behaviour of CRS.

More specifically, as we explained in Section 5.2 at $t=43$ we can derive (DL3) and then from (DL3), the literal L26 in $E_R(44)$ and $A3$ ($A3 \in \text{dep}(\text{dep}((B_{CRS} \cup A_{CRS}) - \{B3\}, B3))$) we can deduce the predicate

$$(DL4) \neg \text{Initiates}(\text{ir:IS:FindAvailable}(ID), \text{equalTo}(v, v2), 42)$$

(DL4) however contradicts the 6th condition of $B3$ which was concerned with the response of the IS service (see literal L27 in the event log of Figure 7) and therefore indicates a case of unjustified behaviour.

As in the example of Section 5.2, this unjustified behaviour is the result of the failure of the SS service to send an `rc:CRS:Depart(ID)` event to CRS. Note, however, that the violation of $A4$ that is detected as an inconsistency of expected behaviour does not indicate that the event L31 that was generated by $B3$ is unjustified. The identification of L31 as an unjustified event is only possible through the detection of the violation of one of the conditions of $B3$.

5.4. Possible Inconsistencies Evidenced from Expected Behaviour

The fourth type of deviations captures possible violations of assumptions and/or behavioural statements of an SBS system. The possibility of such violations is established by events and fluents which are derived from the assumptions specified for an SBS system (as in the case of inconsistencies due to expected behaviour) and possible explanations of events that have occurred in the event log of the system.

Explanations of such events are generated by abductive reasoning using a subset of the set of assumptions of an SBS system that is defined as follows:

Definition 5: Given the set of the behavioural properties B_S and assumptions A_S of an SBS system S , the set of the *abducible assumptions* A_S^{abd} of S is the subset of the formulas $f: B \Rightarrow H$ in A_S that satisfy the following criteria:

- (i) The head H of f includes at least one non negated `Happens` predicate which appears in at least one formula in B_S .
- (ii) The body B of f contains at least one predicate which does not appear in the head of any other formula in A_S . ■

The first condition in Definition 5 is used to ensure that only explanations of events that have been generated during the operation of the system and recorded in the event log of it (as opposed to events that may be assumed by virtue of applying the principle of negation as failure or derived by some assumption) may be used in the process of detecting possible violations. The second condition is used to ensure that explanations of events will always be "minimal" (i.e., as specific as possible¹⁹). The predicates that satisfy condition (ii) of Definition 5 are called "abducible" predicates as in¹⁹. In the case of CRS, for example, only the assumptions $A1$ and $A2$ would be members of the set A_{CRS}^{abd} and could be used to generate explanations of events.

Given a set of assumptions A_S^{abd} that satisfy the above criteria and an event e in the event log of an SBS whose presence cannot be explained by any of the behavioural formulas, our framework generates all the possible explanations of e from the formulas of A_S^{abd} and then checks whether these explanations would – in conjunction with derived and recorded events – result in the violation of a behavioural formula or assumption of the system. Formally, the violations which may be caused by recorded, derived and explanatory predicates are defined as:

Definition 6: Given the set of the behavioural properties B_S and assumptions A_S of a SBS system S , a possible violation of a behavioural property or assumption of S of the form $f:C \Rightarrow A$ occurs at time T if and only if:

$$\{E_R(T), \text{dep}(pForm((B_S \cup A_S) - \{f\}), pForm(f)), \text{dep}(\text{comp}(A_S^{abd} - \{f\}), pForm(f)), EC_a, EC_p\} \models \neg pForm(f)$$

where

- $pForm(f)$ is a formula that is produced from f if all the occurrences of `Happens`, `Initiates`, `Terminates` and `HoldsAt` predicates in f are replaced by the predicates `pHappens`, `pInitiates`, `pTerminates` and `pHoldsAt`. The meaning of these predicates is defined in Figure 8.
- $\text{comp}(A_S^{abd} - \{f\})$ is a set of completing formulas of assumptions in $A_S^{abd} - \{f\}$. The completing formula of a formula $g: B \Rightarrow H$ is defined as a formula $g^c: H \Rightarrow pForm(B)$ where the body B of g is transformed into a $pForm$ and all the constraints of the time variables in the head H of g are transformed into constraints for the time variables in the head $pForm(B)$ of g^c .
- EC_p are additional axioms introduced to reason with the predicates `pHappens`, `pInitiates`, `pTerminates` and `pHoldsAt`. These axioms are defined in Figure 8. ■

In Definition 6, the generation of explanatory events is carried out by deductive reasoning using the completing formulas of the abducible assumptions of an SBS system similarly to the approach of Consolle et al.¹⁹ The difference from their approach is that we generate the completing formula of each of the abducible assumptions in A_S^{abd} by taking the $pForm$ of the body of the original formula. Thus, for example, the completing formula of the assumption $A2$ in Figure 6 is:

$$(A2^c) \quad (\exists t2:\text{Time}, _vID, _PID) \text{Happens}(rc:\text{UI}:\text{RetKey}(_oID2), t2, \mathfrak{R}(t2, t2)) \wedge \\ \text{Initiates}(rc:\text{UI}:\text{RetKey}(_oID2), \text{equalTo}(v, _vID), t2) \wedge \\ \text{Initiates}(rc:\text{UI}:\text{RetKey}(_oID2), \text{equalTo}(p, _PID), t2) \Rightarrow \\ (\exists t1:\text{Time}) \text{pHappens}(rc:\text{SS}:\text{Enter}(_oID1), t1, \mathfrak{R}(t2 - 10t_u, t2 - t_u)) \wedge \\ \text{pInitiates}(rc:\text{SS}:\text{Enter}(_oID1), \text{equalTo}(v, _vID), t1) \wedge \\ \text{pInitiates}(rc:\text{SS}:\text{Enter}(_oID1), \text{equalTo}(p, _PID), t1)$$

The use of the p -counterparts of EC predicates in completing formulas enables the explicit representation of the fact that certain predicates are derived from abducible formulas and therefore they represent uncertain information. It should also be noted that in our framework abducibles are generated without checking whether they preserve the consistency of the original theory (i.e., the behavioural statements and assumptions) as in standard formulations of abductive reasoning^{10, 12, 19}. This is because our objective is not to generate consistent explanations of system events as in pure abductive reasoning but to generate all the possible explanations of these events and then check whether these explanations violate the assumptions or behavioural properties of an SBS system. Finally, it should be noted that, according to Definition 6, abductive reasoning is applied to assumption statements but not to behavioural properties. This is because, behavioural properties are extracted directly from the source code of the composition process of an SBS system and therefore they represent definite sequences of events in the process execution. Thus, if we had a behavioural property $e1 \Rightarrow e2$ and $e2$ had been generated by the execution the path in the SBS composition process which this property represents then $e1$ would have also been recorded in the event log of the system and, therefore, it would not be necessary to generate it by abduction. On the other hand, had $e2$ been produced due to another behavioural property (execution path) of the system the use of $e1 \Rightarrow e2$ in abductive reasoning would not be plausible.

Example. A possible violation of the behavioural statement $B2$ may be detected at $t=43$ given the events of Figure 7 (assuming that $t_0=1$). This is because at this time point, from the literals

L28–L30 which signify the return of the key of the car $v2$ at the car park $p2$ and the assumption $A2$, it is possible to abduce that the car $v2$ entered the car park $p2$ at some time point in the time range $\mathcal{R}(33,42)$ or, equivalently, that an $rc:CRS:Enter(ID)$ event signifying the entrance of $v2$ into $p2$ should have occurred in this time range. However, it is also known that $v2$ had entered the car park $p2$ at $t=18$ (see literals L4–L6 in Figure 7). Thus, from $A1$ it can be deduced that $v2$ should have departed from $p2$ at some time point in the time range $\mathcal{R}(19,41)$ or, equivalently that an $rc:CRS:Depart(ID)$ event signifying this departure should have occurred within $\mathcal{R}(19,41)$. This, however, means that the $depart$ event could have occurred in the sub-range $\mathcal{R}(32,38)$ of $\mathcal{R}(19,41)$ and under this possibility the behaviour of CRS would have violated the formula $B2$. The violation in this case would have occurred since, according to $B2$, following the reply to the car rental request and the allocation of the car $v2$ to it at $t=31$ (as signified by the literal L20 in Figure 7) if the above depart event had occurred in the range $\mathcal{R}(32,38)$, CRS should have called the operation *MakeUnavailable* in the IS service to record the unavailability of $v2$. This operation, however, was not called according to the event log of Figure 7.

p-EC predicates:	
▪	$pHappens(e, t, \mathcal{R}(t1, t2))$: this predicate signifies the possibility that an event e occurs at some time t within the time range $\mathcal{R}(t1, t2)$
▪	$pInitiates(e, f, t)$: this predicate signifies the possibility that an event e initiates fluent f at some time t within the time range $\mathcal{R}(t1, t2)$
▪	$pTerminates(e, f, t)$: this predicate signifies the possibility that an event e terminates fluent f at some time t within the time range $\mathcal{R}(t1, t2)$
▪	$pHoldsAt(f, t)$: this predicate signifies the possibility that a fluent f holds at some time t .
Axioms for p-EC predicates:	
(pEC _i)	($i=1, \dots, 9$) This axiom is the same as axiom EC _i in Figure 2 with the occurrences of each <Predicate> in EC _i except from Initially _p re-written as p<Predicate>
(pEC10)	$(\forall e: Event, t1, t2, t3, t4, t5, t6: Time) Happens(e, t1, \mathcal{R}(t2, t3)) \wedge INT_T(\mathcal{R}(t2, t3), \mathcal{R}(t5, t6)) \neq \emptyset \Rightarrow pHappens(e, t4, INT_T(\mathcal{R}(t2, t3), \mathcal{R}(t5, t6)))$
(pEC11)	$(\forall e: Event, t1, t2, t3, t4, t5, t6: Time) Happens(e, t1, \mathcal{R}(t2, t3)) \wedge INT_T(\mathcal{R}(t2, t3), \mathcal{R}(t5, t6)) \neq \emptyset \Rightarrow pHappens(e, t4, INT_T(\mathcal{R}(t2, t3), \mathcal{R}(t5, t6)))$
where	
$INT_T(RN1, RN2)$ is a function denoting the intersection of two time ranges RN1 and RN2 defined as:	
	$INT_T(\mathcal{R}(t1, t2), \mathcal{R}(t3, t4)) = \mathcal{R}(\max(t1, t3), \min(t2, t4))$ if $\max(t1, t3) \leq \min(t2, t4)$.
	$INT_T(\mathcal{R}(t1, t2), \mathcal{R}(t3, t4)) = \emptyset$ if $\max(t1, t3) > \min(t2, t4)$

Fig. 8. p-Predicates and their axioms.

Formally, the possible violation of $B2$ in this case is detected as follows:

- (i) From the completing formula $A2^c$ of assumption $A2$ and the literals L28, L29 and L30 we can derive the abducibles (AL):
- (AL1) $pHappens(rc:SS:Enter(ID), t, \mathcal{R}(33, 42))$,
- (AL2) $pInitiates(rc:SS:Enter(ID), equalTo(v, v2), t)$
- (AL3) $pInitiates(rc:SS:Enter(ID), equalTo(p, p2), t)$
- (ii) Subsequently, from (AL1)–(AL3), the literals L4–L6 in Figure 7, and the *pForm* of assumption $A1$ (note that $pForm(A1) \in dep(pForm((B_S \cup A_S) - \{B2\}), pForm(B2))$), we can deduce the literals
- (AL4) $pHappens(rc:SS:Depart(ID), t, \mathcal{R}(19, 41))$,

(AL5) $\text{pInitiates}(\text{rc:SS:Depart}(\text{ID}), \text{equalTo}(\text{p}, \text{p2}), \text{t})$

(AL6) $\text{pInitiates}(\text{rc:SS:Depart}(\text{ID}), \text{equalTo}(\text{v}, \text{v2}), \text{t})$

- (iii) From (AL4)–(AL6), however, the literal L20 in Figure 7 and the literal $\neg\text{Happens}(\text{ic:IS:MakeUnavailable}(\text{oID3}, \text{v2}, \text{p2}), \text{t}, \mathcal{R}(31, 38))$ which can be deduced from the event log of Figure 7 by negation as failure we can derive the negation of the *pForm* of B2.

Possible violations as the one in the above example are useful to detect since they may indicate some malfunction of the system or the services deployed by it. In the above example, for instance, the possible violation of B2 may have occurred due to a malfunctioning of the SS service that failed to generate an rc:SS:Depart event in the time range $\mathcal{R}(19, 41)$. And the possibility of the occurrence of this event could not have been determined without the combination of deductive and abductive reasoning that was triggered by the return car key event represented by the literal L28 in Figure 7.

5.5. Possible Cases of Unjustified Behaviour

Abduced explanations of events that have occurred in the event log of an SBS system and cannot be attributed to its behavioural formulas, may also be useful in detecting possible cases of unjustified behaviour. Such cases are defined as follows:

Definition 7: A behavioural statement of the form $f: B \Rightarrow H$ is said to *possibly generate unjustified behaviour* at time T, if and only if there is a literal e such that:

- (i) $e \in E_R(T)$
- (ii) e can be unified with H
- (iii) $\{E_R(T) - \{e\}, B_S, EC_a\} \models e$
- (iv) $\{E_R(T) - \{e\}, B_S - \{f\}, EC_a\} \not\models e$ and
- (v) there is a predicate L in the *pForm* of f for which,
 $\{E_R(T), \text{dep}(\text{pForm}((B_S \cup A_S) - \{f\}), \text{pForm}(f)), \text{dep}(\text{comp}(A_S^{\text{abd}} - \{f\}), \text{pForm}(f)), EC_a, EC_p\} \models \neg \text{pForm}(L)$ ■

Definition 7 is analogous to Definition 4, except that it takes into account not only the extended behaviour of an SBS system but also abduced explanations of this behaviour.

Example. Given the event log in Figure 7, at time $t=38$, the behavioural statement B1 is satisfied by $E_R(38)$. This is due to the literals L20 and L21 and the literal $\neg\text{Happens}(\text{rc:SS:Depart}(\text{ID}), \text{t}, \mathcal{R}(32, 37))$ which can be established from the event log of Figure 6 by the principle negation as failure. However, if the absence of a Depart event in this instance was the result of a malfunctioning SS service or a communication failure between this service and CRS and a Depart event had occurred without being received and recorded by CRS then the behaviour of CRS would have violated B1. This possibility can be detected by applying abductive and deductive reasoning as in the example in section 5.4 and derive the predicates (AL4)–(AL6). These predicates would entail the negation of the *pForm* of the conditions

$$\begin{aligned} & \neg(\exists t2:\text{Time}) \text{Happens}(\text{rc:SS:Depart}(_oID2), t2, \mathcal{R}(t1, t1+6*t_u)) \wedge \\ & \text{Initiates}(\text{rc:SS:Depart}(_oID2), \text{equalTo}(\text{v}, \text{vID}), t2) \wedge \\ & \text{Initiates}(\text{rc:SS:Depart}(_oID2), \text{equalTo}(\text{p}, _pID), t2) \end{aligned}$$

of the behavioural statement B1 at $t=38$ and therefore the event L21 in the event log of the system would constitute a possible case of unjustified behaviour.

6. The Monitoring Process

In this section, we give an overview of the monitoring process. This process is based on algorithms which are discussed in detail in Ref. 9.

At runtime, the monitor maintains templates representing different instantiations of the formulas that specify the behavioural properties and assumptions for an SBS system. A template for a formula f includes:

- The identifier (ID) and type (T) of f . The type of f is F (future) if all the predicates p in f whose time variables are constrained by the time variables of other predicates in the formula must occur after these predicates. The formula $A2$ in Figure 2, for example, is an F formula since the predicates $\text{Happens}(rc:UI:RetKey(_oID2), t2, \mathfrak{R}(t1+t_u, t1+10*t_u))$, $\text{Initiates}(rc:UI:RetKey(_oID2), \text{equalTo}(v, _vID), t2)$ and $\text{Initiates}(rc:UI:RetKey(_oID2), \text{equalTo}(p, _pID), t2)$ whose time variable $t2$ is constrained by the time variable $t1$ of the predicate $\text{Happens}(rc:SS:Enter(_oID1), t1, \mathfrak{R}(t1, t1))$ must occur after the latter predicate. The type of f is P (past) if there is at least one predicate p with a time variable that is constrained by the time variable of another predicate q and p must occur before q . The formula $A1$ in Figure 2, for example, is a past formula since the predicate $\text{Happens}(rc:SS:Depart(_oID3), t3, \mathfrak{R}(t1+t_u, t2-t_u))$, whose time variable $t3$ is constrained by the time variable $t2$ of the predicate $\text{Happens}(rc:SS:Enter(_oID2), t2, \mathfrak{R}(t1+2t_u, t2))$ must occur before the latter predicate.
- A dependants list (DP) of pairs (id, P) which indicate other formulas which depend on f where id is the identifier of a formula that depends on f and p is the predicate that creates the dependency (see Definition 3)),
- The bindings (VB) of the non-time variables of all the predicates in f , and
- For each predicate p in f :
 - The *qualifier* of the time variable (Q) and signature (SG) of p .
 - The *time range* (LB, UB) within which p should occur (the boundaries of this range are determined by constraints for the time variable of p in f).
 - The *truth-value* (TV) of p which can be: UN (if the truth-value of p has not been established), $True$ (if p is true), or $False$ (if p is false).
 - A *time stamp* (TS) indicating the time at which the truth-value of p was established
 - The *source* (SC) of the evidence for the truth value of p which can be: UN (if the truth value of p has not been established), RE (if the truth value of p is established by a recorded literal unified with it), DE (if the truth value of p is established by a derived event unified with it), or NF (if the truth value of p is established by the principle of negation as failure)

The monitor creates three templates for each formula. The first of these templates is used to check for violations of the formula using only recorded predicates. The second template is used to derive predicates from the formula and check for violations of the formula by the expected behaviour and/or cases of unjustified behaviour caused by derived and recorded predicates. And the third template represents the $pForm$ of the formula and is used to check for possible violations of the formula by the expected behaviour and/or possible cases of unjustified behaviour.

The monitor picks recorded events by polling at regular time intervals the *Event Database* of the framework (see Figure 1) and checks if there are instances of templates that should be updated by these events. Updates of a template may be made if a recorded event can be unified with a predicate in the template and the timestamp of the event is within the time range (LB, UB) of the predicate. If a template is updated, the bindings of the variables and the truth value of the relevant

predicate in it are also updated, and its timestamp is set to the timestamp of the recorded event that led to the update. A new instance of a template is generated if a recorded event can be unified with an unconstrained predicate of the template (i.e., a predicate whose time variable is not constrained by the time variable of another predicate) or the variable bindings of the recorded event are different from the variable bindings of a predicate that has the same name with it in the template.

According to this process, following the occurrence of the event signified by the literal L14 in the event log of Figure 7 at $t=28$ the monitor will create the following template instance for formula $B3$:

ID	B3						
T	F						
DP	(A4, Happens(re:UI:CarRequest(_oID3, _vID, _pID)))						
VB	(_oID1, ID11)						
P	Q	SG	TS	LB	UB	TV	SC
1	\forall	Happens(rc:UI:CarRequest(_oID1),t1, $\mathfrak{R}(t1,t1)$)	28	28	28	True	RE
2	\forall	Initiates(rc:UI:CarRequest(_oID1),equalTo(p, _pID),t1)	28	28	28	UN	UN
3	\forall	Initiates(rc:UI:CarRequest(_oID1), equalTo(c, _cID),t1)	28	28	28	UN	UN
4	\exists	Happens(ic:IS:FindAvailable(_oID2, _pID),t2, $\mathfrak{R}(t1,t2)$)	t2	28	t2	UN	UN
5	\exists	Happens(ir:IS:FindAvailable(_oID2),t3, $\mathfrak{R}(t2,t3)$)	t3	t2	t3	UN	UN
6	\exists	Initiates(ir:IS:FindAvailable(_oID2),equalTo(v, _vID),t3)	t3	t3	t3	UN	UN
7	\exists	Happens(re:UI:CarRequest(_oID1, _vID, _pID),t4, $\mathfrak{R}(t3,t3+t_u)$)	t4	t3	t3+1	UN	UN

Subsequently, when the events signified by the literals L15–L20 are encountered, the above template instance will be updated and take the following form:

ID	B3						
T	F						
DP	(A4, Happens(re:UI:CarRequest(_oID1, vID, _pID))						
VB	(_oID1, op4), (_pID, p2), (cID, c2), (_oID2, ID12), (vID, v2)						
P	Q	SG	TS	LB	UB	TV	SC
1	\forall	Happens(rc:UI:CarRequest(_oID1),t1, $\mathfrak{R}(t1,t1)$)	28	28	28	True	RE
2	\forall	Initiates(rc:UI:CarRequest(_oID1),equalTo(p, _pID),t1)	28	28	28	True	RE
3	\forall	Initiates(rc:UI:CarRequest(_oID1), equalTo(c, _cID),t1)	28	28	28	True	RE
4	\exists	Happens(ic:IS:FindAvailable(_oID2, _pID),t2, $\mathfrak{R}(t1,t2)$)	29	29	29	True	RE
5	\exists	Happens(ir:IS:FindAvailable(_oID2),t3, $\mathfrak{R}(t2,t3)$)	30	30	30	True	RE
6	\exists	Initiates(ir:IS:FindAvailable(_oID2), equalTo(v, _vID),t3)	30	30	30	True	RE
7	\exists	Happens(re:UI:CarRequest(_oID1, _vID, _pID),t4, $\mathfrak{R}(t3,t3+t_u)$)	31	31	31	True	RE

Another instance of the template of $B3$ will be created following the occurrence of the event signified by the literal L24 in the event log of Figure 7. This is because L24 cannot be unified with the above instance of the template of this formula.

The status of a predicate in a template instance may also be updated by applying the principle negation as failure. This will happen depending on the quantifier of the time variable of the predicate and its time constraint. Suppose, for instance, that—the event $\text{Happens}(re:UI:CarRequest(ID11, v2, p2), 31, \mathfrak{R}(31, 31))$ in the event log of Figure 7 had not occurred. Then, by negation as failure, at $t=31$ or any time later than it when the first event that does not unify with this predicate occurs the truth value of $\text{Happens}(re:UI:CarRequest(_oID1, _vID, _pID), t4, \mathfrak{R}(t3, t3+t_u))$ would be established as *false* in the second instance of the $B3$ template above. This is because, given the upper time boundary and the existential quantification of the time variable of this predicate, an event unifiable with it

should have occurred up to time $t=31$. Assuming, for example, that the literal L20 had not occurred, this update would have taken place when the monitor encountered the literal L21 in Figure 7 which indicated that the current time in the execution of the composition process is $t=38$.

Updating templates with derived predicates is similar to updating them with recorded predicates. The only difference is that derived predicates might have a time range within which they are known to be true but not a specific timestamp within this range. In such cases, if a derived predicate can be unified with a predicate in a template whose truth value is unknown, the truth value of the template predicate and the template will be updated. This, however, will happen only if the time range of the derived predicate is within the time range of the template predicate. Updates of templates that represent the *pForms* of formulas by abduced predicates take place according to a similar process. In this case, the update is performed only if an abduced predicate can be unified with a template predicate and the time range of the former overlaps with the time range of the latter (see axiom *pEC11* in Figure 8).

Derived and abduced predicates are generated when the truth values of all the predicates in the body of a template have been set to *True* and all the non-time variables of the predicates in the head of a template are unified with concrete values. Then the values of the predicates in the head of the template are set to *True* and bound instances of these predicates are generated.

When the truth values of all predicates in a formula template have been established, a check for possible formula violations is performed according to the criteria described in Ref. 9. In the case of F-formulas, for example, if the truth-value of all the predicates in the template is true the formula is satisfied, and if the truth-value of all the unconstrained predicates in the formula is true and the truth-value of at least one constrained predicate is false and the source of all predicates is *RE* or *NF*, the formula is marked as inconsistent with the recorded behaviour of the system.

7. Evaluation

In this section we describe a series of experiments that we performed to evaluate our monitoring framework and demonstrate its applicability to an SBS system deploying third party web-services. In these experiments we used an implementation of our framework in Java that uses the *bpws4j* engine. The description of this implementation is beyond the scope of this paper and may be found in Ref. 25.

For these experiments we developed a BPEL process, called *Quote Tracker Process (QTP)* which allows a user to get a stock quote in US dollars given a stock symbol from NYSE and convert it to some other currency. *QTP* uses a web service called *Stock Quote Service (SQS)* to get the quote for a NYSE symbol, a second web service called *Currency Exchange Service (CES)* to get the currency exchange rate between the US and a target currency, and a third web service, called *Simple Calculator Service (SCS)*, to convert the quote into the target currency. In our implementation of *QTP*, we used the services *SQS* and *CES* of *XMethods*²⁰ and implemented *SCS* locally.

In the experiments we monitored four different formulas (assumptions): (i) a formula specifying that *CES* should return the same exchange rate for the same country within a fixed time period, (ii) a formula specifying a constraint over the response time of *CES*, and (iii) formulas monitoring the average response time of *SQS*. The specifications of the monitored formulas and the BPEL and WSDL files specifying *QTP* and *SCS* are available at

www.soi.city.ac.uk/~am697/QTP_Case_Study.html. The WSDL file of the *SQS* is available at <http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl> and the WSDL file of the *CES* is available at <http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl>.

The objective of our experiments was to measure the average delay in detecting a deviation. It should be noted that, due to the deployed formulas, the only types of deviations that could occur in the experiments were inconsistencies caused by the recorded and expected behaviour of *QTP*.

	Exp 1	Exp 2	Exp 3	Exp 4
Sleeping interval of QTP client	Uniformly distributed in the range [0,...,10] secs	Uniformly distributed in the range [0,...,5]secs	Uniformly distributed in the range [0,...,10] secs	Uniformly distributed in the range [0,...,5] secs
Event type	Recorded	Recorded	Recorded + Derived	Recorded + Derived
Average inter-arrival event time (s)	1.052	0.278	0.449	0.308
Total event time span in secs/hours	10526.08/~2.92 h)	2781.01/~ 0.77 h	4499.02/~1.25 h	3084.81/~ 0.85 h

Table 1. Experiments summary

To deploy *QTP* over a non trivial period of time and with a non trivial level of load, we developed a client program for it. This program picks up randomly a stock symbol from a set of 15 symbols and calls *QTP* for a quote for this symbol. When it receives the quote from *QTP*, the client program randomly picks up a country name from a set of 15 country names and calls *QTP* to convert the quote in the country's currency. Subsequently, the client program sleeps for a random time interval and then repeats the above sequence of actions with different sets of data. The random sleeping interval of the client program was used in the experiments to control the inter-arrival time of the events exchanged between the deployed web-services and *QTP*. In total, we carried out four experiments by varying the sleeping interval of the *QTP* client and the types of the used events. The settings of these experiments are summarized in Table 1.

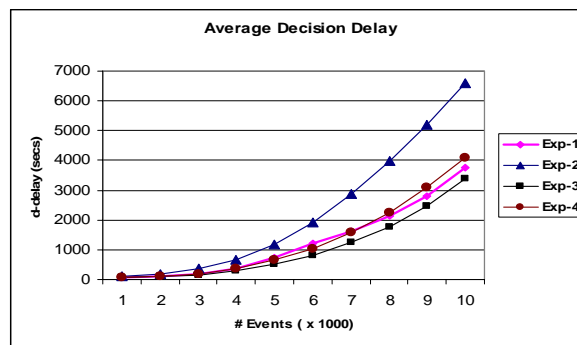


Fig. 9. Average decision delay

In each experiment, we computed the average delay in making a decision about possible violations of a formula, called *d-delay*, using the formula

$$d\text{-delay} = \sum_{j=1, \dots, N}^{F_j} (T_E^{F_j} - \max_{i \in F_j} (t_i^{e(d)})) / N$$

where $t_i^{e(d)}$ is the time of recording the event i in the monitor's database, $T_E^{F_j}$ is the completion time of the decision procedure that the monitor executes to check for deviations given the truth values

of the predicates in the template j for a formula F , and N is the number of the formula templates for which a decision was made during an experiment.

The d-delay for detecting different types of deviations is shown in Figure 9. As shown in this figure, d-delay ranged from 1.8 hours in the case of *Exp-2* (i.e., the experiment with the fastest event arrival pace in which 10,000 events were received within 0.77 hours) to 0.99 hours in the case of *Exp-3*. In all experiments we observed a consistent pattern of an exponential increase in d-delay along with increasing event numbers. Also the results for experiments *Exp-2* and *Exp-3* suggest that the use of recorded or mixed event sets did not affect the average delay in the detection of deviations. This was due to the fact that the monitoring of mixed events was happening in parallel with the monitoring of recorded events. This result confirms a similar observation that we made whilst evaluating an earlier version of the framework using a simulated BPEL process²¹.

The above results demonstrate that the average delay in the detection of a requirement deviation might not be negligible. Decision delays may limit the applicability of our framework to monitoring only certain types of properties where the timeliness in the detection of a deviation is not critical for a system (e.g., monitoring of long term performance properties of a system) and exclude time critical properties (e.g. safety). Regardless, however, of the properties being monitored it should be appreciated that since monitoring takes place in parallel with the operation of an SBS system without affecting its performance, it can detect useful deviations from requirements at no significant cost for the system.

8. Related Work

Run-time requirements monitoring has been the focus of different strands of requirements engineering research (e.g. see Refs. 1-3) which have investigated: (i) ways of specifying requirements for monitoring and transforming them into events that can be monitored at run-time; (ii) the development of event-monitoring mechanisms; (iii) the development of mechanisms for generating system events that can be used in monitoring; and (iv) the development of mechanisms for adapting systems in order to deal with deviations from requirements at run-time.

Most of the existing techniques^{12,13} express requirements in the KAOS framework¹⁴ that provides a goal oriented formal specification language based on temporal logic. In this approach, requirements are initially specified as high level goals that must be achieved by a system. These goals are refined and mapped onto events that can be monitored at run-time. This transformation is the responsibility of system providers who may apply goal refinement patterns (as in Ref. 2), or map requirements onto operations defined in an intermediate design model, which can then be mapped onto monitorable events³. To monitor the existence of specific patterns of events that indicate the violation of requirements, the existing techniques use either special purpose monitoring architectures (e.g. AMOS¹⁵ and FLEA¹²) that maintain event logs and offer proprietary event pattern specification languages, or store events in relational databases and deploy standard SQL querying for detecting requirement violations.¹³

Typically, 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*). These statements may be inserted in the source or compiled code of a system (see Refs. 16 and 3, respectively). The main drawback of instrumentation is that it has to be done manually. To

alleviate this problem, Dingwall-Smith and Finkelstein¹⁶ have developed an aspect oriented approach, in which system providers specify instrumentation code in separate classes, and define composition rules that determine how this code is to be merged with application code. Capra et al⁴ have also suggested an alternative approach based on the use of reflective middleware that can maintain metadata about an application and its execution context, and give dynamic access to this information upon request.

In Ref.17, requirements are expressed as a finite state automaton that models the acceptable system behaviour. In this approach, at run time the conditions that determine the presence of a specific state and the validity of different transitions from it are evaluated against event occurrences in order to establish if the observed system behaviour is compliant with the automaton.

More recently, there has been research focusing on monitoring of SBS systems. Baresi et al²⁴, for example, have developed a tool that instruments the composition process of an SBS system specified in BPEL in order to make it call external monitoring services that check assertions at runtime. Instrumentation is driven by annotations of the BPEL process that specify the assertions to be checked. The execution of the composition process waits until the monitor service returns the result of the check and it may continue or raise an exception depending on whether the assertion has been violated. Thus, Baresi's approach does preventive monitoring unlike ours. However, our approach is not intrusive to the operation of an SBS system and the monitoring that it can perform does not affect the performance of this system. Furthermore, our approach makes it possible to monitor more than one properties not in isolation (as Baresi's approach does) but jointly.

Our requirements monitoring framework builds upon techniques developed for integrity checking in temporal¹⁸ and temporal deductive databases.⁷ The main difference from these techniques is that the formula checking scheme that we use can – in addition to classic integrity constraints violations which correspond to inconsistencies evidenced from recorded and expected behaviour in our framework – detect cases of unjustified behaviour, possible inconsistencies of expected behaviour and possible cases of unjustified behaviour.

9. Conclusion

In this paper, we have presented a framework for monitoring requirements for service-based software systems. These requirements may take the form of behavioural properties or assumptions of such systems, their constituent services and the behaviour of agents interacting with them. The behavioural properties are initially extracted from the specification of the composition process of an SBS system that is expressed in BPEL. This ensures that the properties to be monitored are expressed in terms of events occurring during the interaction between the composition process and the constituent services of the system and, therefore, can be detected at runtime. The assumptions to be monitored are subsequently defined in terms of these detectable events. The specification of assumptions is supported by a graphical editor which allows users to select detectable events and define formulas over them. Both the behavioural properties and assumptions are specified in event calculus.

The framework supports the monitoring of five different types of deviations, namely (i) inconsistencies evidenced from recorded behaviour, (ii) inconsistencies evidenced from expected behaviour, (iii) unjustified system behaviour, (iv) possible inconsistencies evidenced from

expected behaviour, and (v) possibly unjustified system behaviour. According to our experience, these deviations capture a broad spectrum of situations where the behaviour of a service based system at run-time may violate expectations. In particular, it should be noted that deviations of the former two types are supported by other runtime monitoring systems. Cases of unjustified behaviour are also useful as they reveal SBS system actions that would not be taken had the constituent services of the system functioned according to expectations. Also deviations of the types (iv) and (v) capture possible cases of problematic behaviour which can be revealed through the generation of explanations of the behaviour of a system by abductive reasoning. Such explanations are particularly useful in cases where the absence of run-time events may be due to malfunctioning of system components (e.g. failure to report information). Overall, however, it cannot be guaranteed there are no other types of deviations that could be interesting to observe at run-time.

Currently, we are focusing on a large scale experimental evaluation of the framework in order to establish the effect of different factors such as the size of the domains of non time variables in the average performance of the framework. We are also working on the development of support for using the framework to monitor and modify service level agreements. This work investigates the possibility of integrating our framework with existing standards (e.g., WS-Agreement²⁶) and defining a higher level language for specifying monitorable requirements based on the formal semantics of event calculus language that we have presented in this paper. This direction is significant as it could make it easier for users who are not familiar with event calculus to deal with the complexity of specifying monitorable requirements in it.

Acknowledgements

The work reported in this paper has been partially funded by the European Commission under the Information Society Technologies Programme as part of the project SeCSE (contract IST-511680).

References

1. Feather M., Fickas S. "Requirements Monitoring in Dynamic Environments". Proc. of 2nd IEEE Int. Symposium on Requirements Engineering, p. 140, 1995
2. Feather M.S., Fickas S., Van Lamsweerde A. and Ponsard C. "Reconciling System Requirements and Runtime Behaviour". Proc. of 9th Int. Workshop on Software Specification & Design, p. 50, 1998.
3. Robinson W. "Monitoring Software Requirements using Instrumented Code". Proc. of the 35th Hawaii Int. Conf. on Systems Sciences, p. 276b, 2002.
4. Capra L., et al. "Reflective middleware solutions for context-aware applications", LNCS 2192, p. 126, 2001
5. Andrews T. et al. "Business Process Execution Language for Web Services", v1.1, <http://www-106.ibm.com/developerworks/library/ws-bpel>
6. Shanahan M. "The event calculus explained", In Artificial Intelligence Today, LNCS: 1600, p. 409, 1999
7. Plexousakis D. "Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases", Proc. of the 19th Int. Conf. on Very Large Data Bases, p. 146, 1993
8. BPWS4J, <http://alphaworks.ibm.com/tech/bpws4j>
9. Mahbub K., Spanoudakis G. "A Scheme for Requirements Monitoring of Web Service Based Systems", Technical Report Series, Department of Computing, City University, London, 2004
10. Eiter T., Gottlob G. "The complexity of logic-based abduction", *Journal of the ACM*, 42(1), p. 3. 1995.
11. Eshghi K., Kowalski R. "Abduction through Deduction". Technical Report, Imperial College of Science & Technology, Department of Computing, 1988.

12. G. Paul. "Approaches to Abductive Reasoning: an overview", *Artificial Intelligence*, 7, p. 109, 1993.
13. Robinson W.N., "Monitoring Web Service Requirements", *Proc. of 11th Int. Conf. on Requirements Engineering*, p. 65, 2003
14. Dardenne A., et al. "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, 20, p. 3, 1993.
15. Cohen D. et al. "Automatic Monitoring of Software Requirements". *Proc. of the 19th Int. Conf. on Software Engineering*, p. 602, 1997
16. Dingwall-Smith A., Finkelstein A. "From Requirements to Monitors by Way of Aspects". *Proc. of 1st Int. Conf. on Aspect-Oriented Software Development*, 2002
17. Peters D. K.. "Deriving Real-Time Monitors from System Requirements Documentation". *Proc. of 3rd Int. Symposium on Requirements Engineering, Doctoral Consortium*, 1997
18. Chominski J. "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding". *ACM Transactions on Database Systems*, 20(2), p. 149, 1995
19. Console L., Dupre D. T., and Torasso P., "On the Relationship between Abduction and Deduction". *Journal of Logic and Computation*, 1(5), p. 661, 1991
20. XMethods, <http://www.xmethods.net/>
21. Mahbub K, Spanoudakis G. "Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience". *Proc. of IEEE Int. Conf. on Web Services (ICWS'05)*, 257-265, 2005.
22. Mahbub K., Spanoudakis G. "A framework for Requirements Monitoring of Service Based Systems". *Proc. of 2nd Int. Conf. on Service Oriented Computing*, p. 84, 2004.
23. Oracle, "Orchestrating Web Services: The Case for a BPEL Server". An Oracle White Paper, June 2004 (available from http://www.oracle.com/appserver/bpel_home.html on 17/12/2005)
24. Baresi L., Ghezzi C., and Guinea S. "Smart Monitors for Composed Services". *Proc. of 2nd Int. Conf. on Service Oriented Computing*, p. 193, 2004
25. Mahbub K., "A4: City Monitoring Tool – User Manual". Technical Report, SECSE project, 2005 (available from: www.soi.city.ac.uk/~gespan/secse/CityMonitoringTool_UserManual2005.pdf)
26. 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>