Run-time Monitoring of Requirements for Systems Composed of Web-Services: Initial Implementation and Evaluation Experience

Khaled Mahbub and George Spanoudakis
Department of Computing, City University,
London EC1V 0HB, United Kingdom
E-mail: {am697 | gespan}@soi.city.ac.uk

Abstract

This paper describes a framework supporting the run-time monitoring of requirements for systems implemented as compositions of web-services specified in BPEL. The requirements that can be monitored are specified in event calculus. The paper presents an overview of the framework and describes the architecture and implementation of a tool that we have developed to operationalise it. It also presents the results of a preliminary experimental evaluation of the framework.

1. Introduction

Run-time requirements monitoring is the activity of checking whether at run-time a software system operates according to requirements set for it [5], [6]. This form of monitoring is required to detect violations of requirements that cannot be detected by static verification (e.g. model checking). Violations of requirements may not be detectable by static verification if: (a) the satisfaction of these requirements depends on assumptions about the behaviour of actors in a system's environment that cannot be verified before the system is put in operation (see assumption A1 in Section 2 for example), or (b) system specification models are incomplete or have an infinite or large number of states that makes static verification intractable.

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

In this paper, we present a framework that we have implemented to support the run-time verification of requirements for SBS systems and discuss the results of a preliminary experimental evaluation of it. The formal foundations of this framework are discussed in [13].

Our framework supports the run-time monitoring of *behavioural properties* of an SBS system or *assumptions* about the behaviour of the different web-services that constitute it or agents in its environment. Behavioural properties are automatically extracted from the specification of the composition process of the SBS system which our framework assumes to be expressed in BPEL [1]. Assumptions are additional requirements about the behaviour of agents interacting with the system, or the individual services of it. Assumptions are specified by system providers in *event calculus (EC)* [10] using an XML schema that we have developed to support the representation of EC formulas.

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

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

The rest of the paper is structured as follows. In sections 2 and 3, we present the foundations of our run-time monitoring framework and an overview of its architecture, respectively. In Section 4, we discuss its monitoring process. In Section 5, we present the results of an initial experimental evaluation of the framework. In Section 6, we overview related work and in Section 7 we summarise our approach and plans for future work on it.

2. Foundations of Run-time Monitoring

2.1 Specification of Requirements

In our framework, the behavioural properties and assumptions that are to be monitored at run-time are

Proceedings of the IEEE International Conference on Web Services (ICWS'05)
0-7695-2409-5/05 $20.00 **IEEE**

specified using an XML schema that represents formulas of *event calculus* (EC) [10]. Event calculus is a first-order temporal logic language that can be used to specify the *events* that occur within a system and the effects of these events (called *fluents* in EC).

In an SBS system, the events that may occur can be of 5 different types:

(i) *Invocation events* that signify the invocation of an operation in one of the partner services of an SBS system by its composition process. These events are represented by terms of the form: *ic:Service:OperationName(Parameters)*

(ii) *Return events* that signify the return from the execution of an operation that was invoked by the composition process of an SBS system in one of its partner services. These events are represented by terms of the form: *ir:Service:OperationName(Parameters).*

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

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

(v) *Assignment events* that signify the assignment of a value to a variable in the composition process of an SBS system. Assignment events are represented in our framework by terms of the form: *as:AssignmentName(assignmentId)*

The occurrence of an event is represented by the predicate *Happens(e,t,*$\Re(t_1,t_2)$*)* which signifies that an event *e* occurs at some time *t* within the time range $\Re(t_1,t_2)$. The boundaries of $\Re(t_1,t_2)$ can be specified by using either time constants, or arithmetic expressions over the time variables of other *Happens* predicates of the same formula.

An event may initiate or terminate a fluent. A fluent is specified as a condition over the value of a specific variable of the composition process of a system. The fluent *equalTo(x,y)*, for example, signifies that the value of the variable *x* is equal to *y*. The effects of events on fluents are represented by the predicates *Initiates(e,f,t)* and *Terminates(e,f,t)*. *Initiates(e,f,t)* signifies that a fluent *f* starts to hold after the event *e* at time *t*. *Terminates(e,f,t)* signifies that a fluent *f* ceases to hold after the event *e* occurs at time *t*. An EC formula may also use the predicate *HoldsAt(f,t)* which signifies that the fluent *f* holds at time *t*.

An EC formula in our framework can also specify additional constraints about the time variables of predicates using the predicates < and = (*t1* < *t2* is true if *t1* is a time instance that occurred before *t2*, and *t1* = *t2* is true if *t1* is a time instance that is equal to *t2*).

Behavioural properties:

B1. $(\forall$ t1:Time) $(\exists$t2:Time) $(\exists$t3:Time)
Happens(rc:UI:CarRequest(oID1),t1,$\Re$(t1,t1)) $\wedge$
Initiates(rc:UI:CarRequest(oID1), equalTo(p,pID),t1) $\wedge$
Happens(ic:IS:FindAvailable(oID2, pID),t2, $\Re$(t1,t2)) $\wedge$
Happens(ir:IS:FindAvailable(oID2),t3, $\Re$(t2,t3)) $\wedge$
Initiates(ir:IS:FindAvailable(oID2), equalTo(res,vID),t3) $\Rightarrow$
($\exists$t4:Time) **Happens**(re:UI:CarHire(oID3,vID), t4, $\Re$(t3,t3+t_u))

Assumptions:

A1. $(\forall$ t1,t2:Time) **Happens**(rc:SS:Enter(oID1),t1,$\Re$(t1,t1)) $\wedge$
Initiates(rc:SS:Enter(oID1), equalTo(v1, vID), t1) $\wedge$
Initiates(rc:SS:Enter(oID1), equalTo(p1, pID1), t1) $\wedge$
Happens(rc:SS:Enter(oID2),t2,$\Re$(t1+t_u,t2)) $\wedge$
Initiates(rc:SS:Enter(oID2), equalTo(v2, vID), t2) $\wedge$
Initiates(rc:SS:Enter(oID2), equalTo(p2, pID2), t2) $\Rightarrow$
($\exists$t3:Time) **Happens**(rc:SS:Depart(oID3),t3,$\Re$(t1+t_u,t2−t_u)) $\wedge$
Initiates(rc:SS:Depart(oID3), equalTo(v3, vID), t3) $\wedge$
Initiates(rc:SS:Depart(oID3), equalTo(p3, pID1), t3)

A2. $(\forall$ t1, t2:Time)
Happens(ic:IS:FindAvailable(oID,pID), t1, $\Re$(t1,t1)) $\wedge$
Happens(ir:IS:FindAvailable(oID), t2, $\Re$(t1,t2)) $\wedge$
HoldsAt(equalTo(availability(vID1),"not avail"), t2−t_u) $\Rightarrow$
$\neg$**Initiates** (ir:IS:FindAvailable (oID),equalTo(vID2, vID1), t2)

A3. $(\forall$ t1, t2, t3:Time)
Happens(ic:UI:RelKey(oID1,vID), t1,$\Re$(t1, t1))
Happens(ir:UI:RelKey(oID1), t2,$\Re$(t1, t2)) $\wedge$
Happens(rc:UI:RetKey(oID2), t3,$\Re$(t2, t3)) $\wedge$
Initiates(rc:UI:RetKey(oID2), equalTo(v, vID), t3)
$\Rightarrow$ ($\forall$t4:Time) ((t1 < t4)$\wedge$ (t4 < t3)
HoldsAt(equalTo(available(vID),"not-avail"),t4)

In the above formulas, all non-time variables are assumed to be universally quantified and t_u is the minimum time between two events.

Figure 1. Requirements for CRS

Figure 1 shows some examples of behavioural properties and assumptions for a car rental system (CRS). CRS acts as a broker offering its customers the ability to rent cars provided by different car rental companies, directly from car parks at different locations. CRS interacts with:

- *Car information services* (IS) provided by different car rental companies to maintain registries of cars, check car availability and allocate cars to customers.
- *Sensoring services* (SS) provided by different car parks in order to sense cars as they enter in or depart from car parks, and inform CRS accordingly.
- *User interaction services* (UI) handling interactions with end-users.
- A *Payment service* (PS) that CRS uses to take electronic payments for car rentals.

In a typical scenario, CRS receives a car rental request from a UI service 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 through the PS service. When cars move in and out of car parks, respective SS services inform CRS, which subsequently invokes operations in IS services to update the availability status of the moved car.

The formula *A1* in Figure 1 expresses an assumption about the behaviour of the sensoring services (SS) of CRS. According to it, if a car *vID* is sensed to enter a car park *pID1* at some time *t1* and later at time *t2* the same

car is sensed to enter the same or a different car park, then a *Depart* event signifying the departure of *vID* from *pID1* must have also occurred between the two enter events. 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 *Initiates* predicates in the same formula initiate fluents that represent the specific value bindings of the input parameters v_i and p_i (i=1,…,3) of the operations *Enter* and *Depart*. *A1* represents a composite requirement whose satisfiability depends on the availability of SS services and their ability to function correctly. This requirement cannot be verified by static analysis and must be monitored at run-time.

2.2 Deviations

As discussed in Section 1, our framework can detect violations of assumptions by the recorded run-time behaviour of a system, violations of assumptions and/or behavioural properties by the expected behaviour of a system, and cases of unjustified behaviour.

Violations of assumptions by recorded behaviour. As defined in [13], an assumption f is violated by the recorded behaviour of a system at time *T* if the negation of *f* is entailed by the set of the recorded events $E_R(T)$ that have been produced by the system until *T* or, formally, if: $\{E_R(T)\} \models_{nf} \neg f$ ($\models_{nf}$ signifies entailment using the normal rules of inference of first-order logic and the principle of negation as failure).

L1 : Happens(rc:SS:Enter(op1),1,ℜ(1,1))
L2 : Initiates(rc:SS:Enter(op1), equalTo(v1,veh1),1)
L3 : Initiates(rc:SS:Enter(op1), equalTo(p1,loc1),1)
L4 : Happens(rc:SS:Enter(op2),27,ℜ(27,27))
L5 : Initiates(rc:SS:Enter(op2), equalTo(v1,veh1),27)
L6 : Initiates(rc:SS:Enter(op2), equalTo(p1,loc3),27)
L7 : Happens(ic:UI:RelKey(op3, veh2),28, ℜ(28,28))
L8 : Happens(ir:UI:RelKey(op3), 29, ℜ(29,29))
L9 : Happens(rc:UI:CarRequest(op4),49, ℜ(49,49))
L10 : Initiates(rc:UI:CarRequest(op4),equalTo(p,loc2),49)
L11 : Happens(ic:IS:FindAvailable(op5,loc2),50, ℜ(50,50))
L12 : Happens(ir:IS:FindAvailable(op5), 51, ℜ(51,51))
L13 : Initiates(ir:IS:FindAvailable(op5), equalTo(Res,veh2),51)
L14 : Happens(re:UI:CarHire(op6,veh2,loc2), 52, R(52,52))
L15 : Happens(rc:SS:Enter(op7),53,ℜ(53,53))
L16 : Initiates(rc:SS:Enter(op7), equalTo(v1,veh2),53)
L17 : Initiates(rc:SS:Enter(op7), equalTo(p1,loc4),53)
L18 : Happens(rc:UI:RetKey(op8),54,ℜ(54,54))
L19 : Initiates(rc:UI:RetKey(op8), equalTo(v, veh2), 54)
L20 : Happens(rc:UI:CarRequest(op9),69, ℜ(69,69))

Figure 2. Event log of CRS

Assuming the log of events of the CRS system shown in Figure 2, the recorded behaviour of CRS violates the assumption *A1*. This is because there are two enter events that signify the entrance of *veh1* first to car park *loc1* at T=1 (see literals L1-L3 in Figure 2) and, subsequently, to car park *loc3* at T=27 (see literals L4-L6 in Figure 2) but no depart event to signify the departure of *veh1* from *loc1* between these enter events.

Violations of assumptions and/or behavioural properties by expected behaviour. A behavioural property or assumption of the form f: $C1 \Rightarrow A1$ is violated by the expected behaviour of an SBS system if the negation of *f* is entailed by the set of the recorded events of the system and the events that can be deduced from them by the behavioural properties and assumptions of the system that *f* depends on. As defined in [13], *f* depends on a formula g: $C2 \Rightarrow A2$ if the head *A2* of *g* has a literal *L* that unifies either with some literal *K* in the body *C1* of *f* or with some literal *K* in the body of another formula *h* that *f* depends on. Assuming that $dep(f)$ is the set of the formulas that *f* depends on and $E_U(dep(f),T)$ is the set of events that can be produced from these formulas by deduction, a behavioural property or assumption *f* is violated by the expected behaviour of an SBS system at time T if: $\{E_R(T), E_U(dep(f),T), EC_a\} \models_{nf} \neg f$ (EC_a denotes the standard set of axioms of event calculus [10]).

Given the event log of Figure 2, the assumption *A2* is violated by the expected behaviour of CRS. *A2* is an assumption about the behaviour of *IS* services. According to this assumption, the operation *FindAvailable*, which is provided by the IS service of CRS and searches for available cars at specific car parks should not return the identifier of a car to CRS unless this car is available. The violation of *A2* in this case occurs since from the assumption *A3* in Figure 1 (*A3* states that whilst a customer has the key of a car this car cannot be available for renting) we can derive that *veh2* could not be available from T=30 when its key was released (see literals L7 and L8 in Figure 2) until T=53 (that is one time unit before its key was returned back - see literals L18 and L19 in Figure 4). Nevertheless, the execution of the operation *FindAvailable* of the IS service at T=51 reported *veh2* as an available vehicle (see literal L13 in Figure 2).

Unjustified Behaviour. The third type of deviation that can be detected by our framework occurs when the conditions of a behavioural property f that has generated an event *e* are satisfied by the recorded system behaviour but violated by the expected system behaviour. In such cases, the generation of the event *e* is the result of wrong assumptions about the satisfiability of the conditions of f that the system makes at run-time, and constitutes what we refer to as "unjustified behaviour". Formally, a behavioural property of the form f: $C \Rightarrow A$ is said to generate *unjustified behaviour* if and only if there is a literal *e* such that

(i) $e \in E_R(T)$ and *e* can be unified with A
(ii) $\{E_R(T) - \{e\}, f, EC_a\} \models_{nf} e$ and $\{E_R(T) - \{e\}, B_S - \{f\}, EC_a\} \not\models_{nf} e$ (B_S is the set of the behavioural properties of a system)
(iii) there is a literal L in C for which, $\{E_R(T), E_U(dep(f),T), EC_a\} \models_{nf} \neg L$

The conditions (i)-(ii) identify an event *e* which has been generated by the system due to the realisation of a

formula f. The satisfaction of these conditions implies that the conditions of f are satisfied by the recorded behaviour of the system. Note, however, that according to condition (iii), there is some condition in C that would not be satisfied if all the events that could be generated by formulas which f depends on are taken into account. In such cases, e is the result of behaviour that is based on wrong assumptions about the satisfiability of the conditions of f that the system makes at run-time.

Given the event log of Figure 2, a case of unjustified behaviour of CRS that has been caused by the behavioural property *B1* can be detected at T=54. *B1* states that following the receipt of a request for a car rental, CRS will contact IS services to find an available vehicle and if such a vehicle can be found it accept the request. More specifically in this case, as the literals L9-L13 in Figure 2 indicate, all the conditions of *B1* were satisfied at T=51 and therefore CRS replied to the car hire request that it had received from its UI service by invoking the operation *CarHire* in it at *T=52* (see the literal L14 in Figure 2) as specified by *B1*. Note, however, that if the IS and SS services of CRS had behaved according to the assumptions *A2* and *A3* respectively the condition *Initiates(ir:IS:FindAvailable(oID2), equalTo(res,vID),t3)* of *B1* would have been violated. The violation of this condition of *B1* can be deduced from:

- the literals L11 and L12 the event log of Figure 2;
- the assumption *A2* about the behaviour of SS (*A2* belongs to *dep(B1)*), and
- the literal *HoldsAt(equalTo(availability(veh2), "not avail"), 50)* that can be derived from the literals L7, L8, L18 and L19 in the event log of Figure 2 and the assumption *A3* (*A3* belongs to *dep(A2)*).

In other words, if IS and SS had behaved as expected by the assumptions *A2* and *A3* in this case, *veh2* should not have been reported by the operation *FindAvailable* as available and, subsequently, *veh2* should not have been hired. The formal derivation of the inconsistency in this example is discussed in [13].

3. Architecture of the Framework

Our monitoring framework has been implemented in Java™ and incorporates the components shown in Figure 3, namely: a *behavioural properties extractor*, an *assumption editor,* an *event receiver*, a *monitor*, and a *deviation viewer*.

The *behavioural properties extractor* extracts the behavioural properties to be monitored from the BPEL composition process of an SBS system. Behavioural properties are extracted according to the patterns that we describe in [13] and represented in an XML-based language that we have defined to represent EC formulas. This language cannot be presented here due to limited space but is discussed in [12]. The properties extractor also identifies events, effects and state variables in the SBS composition process that provide the primitive constructs for specifying further assumptions about the behaviour of the system. These assumptions are specified by system providers using the *assumption editor*.

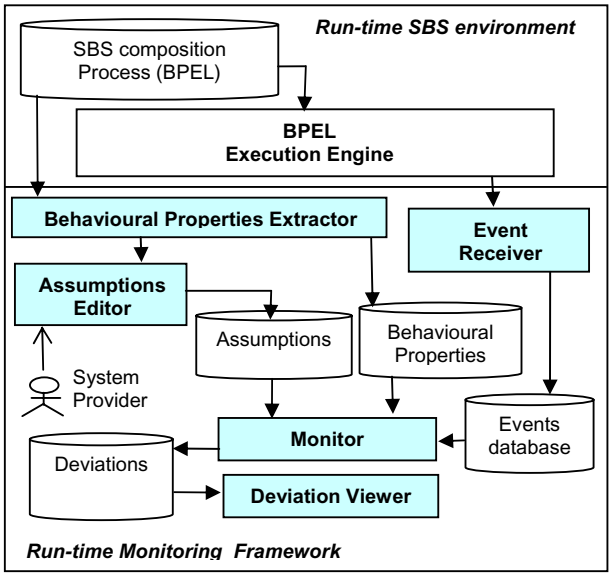

Figure 3. Monitoring framework

The assumption editor presents system providers with the different types of events and fluent initiation predicates that have been identified in the SBS composition process and supports the specification of assumptions as logical combinations of these event and fluent initiation predicates. System providers may also use the editor to define additional fluents to represemt service and system states and relevant initiation and holding predicates. When an assumption is specified, the assumption editor can check its syntactic correctness. Figure 4 presents an intermediate step in specifying the assumption *A1* using the assumption editor.

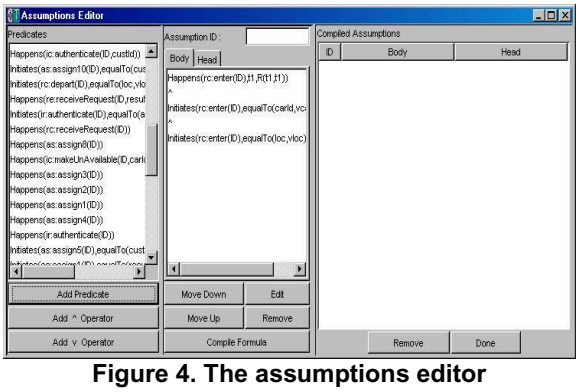

Figure 4. The assumptions editor

While executing the composition process of an SBS system, the process execution engine generates events which are sent as string streams to the *event receiver* of our framework. In our implementation, we have used the

bpws4j process execution engine [8] that uses *log4j* [9] to generate logs of the events during a BPEL process execution. The event receiver identifies the type of the events in its input stream, filters out events which are irrelevant to the monitoring process and records all other events in an *event database*. Irrelevant events are determined by the formulas that have been extracted or specified for monitoring by the system provider.

The *monitor* processes the events which are recorded in its database by the event receiver in the order of their occurrence, identifies other expected events that should have happened but have not been recorded (these events are derived from the assumptions by deduction), and checks if the recorded and expected events are compliant with the behavioural properties and assumptions of the system. In cases where the recorded and expected events are not consistent with these requirements, the monitor records the *deviation* in a database.

The framework incorporates also a *deviation viewer* that is used to browse the detected violations of the formulas. A snapshot of this viewer is shown in Figure 5.

4. The Monitoring Process

At runtime, the monitor maintains templates that represent different instantiations of the formulas to be checked. A template for a formula *f* stores:

- The identifier (*Id*) and type of *f*. The type of *f* is *F* (future) if all the predicates in *f* whose time variables are constrained by time variables of other predicates must occur after these predicates (e.g., formula *B1* in Figure 1) or *P* (past) if there is one predicate *p* that must occur before another predicate *q* that constraints it (e.g., formula *A1* in Figure 1).
- A list of pairs *(i, p)* indicating formulas depending on *f* (*i*) and the predicate creating the dependency (*p*).
- The current unification u_t computed for the template.
- For each predicate *p* in *f*:

- The *quantifier* of its time variable (Q) and its signature (SG).
- The boundaries (*LB*, *UB*) of the time range in which *p* should occur.
- The *truth-value (V)* of *p* which can be: *UN* (i.e., unknown), *T* (true), or *F* (false).
- The *source* (SC) of the evidence for the truth value of *p* which can be: *UN* (if the truth value has not been established), *RE* (if the truth value is established by a recorded event), *DE* (if the truth value is established by a derived event), or *NF* (if the truth value is established by the principle of negation as failure)
- A *time stamp* (*TS*) indicating the time in which the truth-value of *p* was established.

The monitor picks events in the order of their occurrence from the event database and checks if there are formula templates that should be updated by them. A template is updated by an event as specified in the algorithm shown in the appendix. This algorithm distinguishes two types of predicates: (i) predicates with existentially quantified time variables and (ii) predicates with universally quantified time variables.

***Existentially quantified predicates*.** The truth-value of a predicate of the form $(\exists t)p(x,t)$ where *t* must be in the range $\Re(t1,t2)$ is set to *true* as soon as the first event *e* that can be unified with *p* occurs between *t1* and *t2*. If no such event occurs at the distinguishable time points between *t1* and *t2* the truth value of *p* is set to *false*. The absence of events unifiable with *p* is confirmed as soon as the first event that cannot be unified with *p* occurs either on *t2* or after this time point. The truth value of a predicate of the form $\neg(\exists t)p(x,t)$ is established in the opposite way: as soon as an event *e* that can be unified with *p* occurs between *t1* and *t2* the truth value of *p* is set to *false* and if no such events occur at the distinguishable time points between *t1* and *t2* it is set to *false*.

***Universally quantified predicates*.** The truth value of a predicate of the form $(\forall t)p(x,t)$ where *t* must be in the range $\Re(t1,t2)$ is set to *false* as soon as an event that is not unifiable with *p* occurs between *t1* and *t2*, and to *true* if all the events that occur at the distinguishable time points between *t1* and *t2* can be unified with *p*. For predicates of the form $\neg (\forall t)p(x,t)$, the algorithm sets their truth value to *true* as soon as the first event that is not unifiable with *p* occurs within the time range $\Re$(t1,t2) and *false* if no such event occurs at any of the distinguishable time points between *t1* and *t2*. A special kind of universally quantified predicates are unconstrained predicates of the form $(\forall t)p(x,t)$ whose time range is instantaneous and is not constrained by the time variables of other predicates (i.e., a range of the form $\Re(t,t)$). The truth value of such predicates is set to *true* as soon as an event that can be unified with them is encountered by the monitor.

When the truth values of all predicates in a template have been determined, a check for possible formula violations is performed. 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. If the truth value of all the predicates in the body B of a formula f: $B \Rightarrow H$ is *true* and the truth-value of at least one predicate in the head *H* of it is *false* and the source of all predicates is *RE* or *NF*, a violation of the formula by the recorded behaviour of the system is detected.

Example. Following the occurrence of the event L9 in Figure 2 that can be unified with the unconstrained predicate $Happens(rc:UI:CarRequest(oID1),t1,\Re(t1,t1))$ of *B1*, the monitor creates the template shown below for this formula. The truth value (V) of the predicate $Happens(rc:UI:CarRequest(oID1),t1,\Re(t1,t1))$ in this template is set to true, its time stamp (TS) along with the upper and lower bound of the time range of the predicate (UB and LB) are set to 49 (i.e., the time stamp of the event L9), and the source (SC) of the truth value of the predicate is set to *RE* to signify the update of the truth value due to a recorded event.

Id:		B1	T	F	DP			
u_t		{ (oID1, op4) }						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**V**	**SC**	
1	∀	**Happens**(rc:UI:CarRequest(oID1),t1,ℜ(t1,t1))	49	49	49	T	RE	
2	∀	**Initiates**(rc:UI: CarRequest(oID1), equalTo(p,pID),t1)	49	49	49	UN	UN	
3	∃	**Happens**(ic:IS:FindAvailabl e(oID2,pID), t2, ℜ(t1,t2))	t2	49	t2	UN	UN	
4	∃	**Happens**(ir:IS: FindAvailable(oID2),t3, ℜ(t2,t3))	t3	t2	t3	UN	UN	
5	∃	**Initiates**(ir:IS: FindAvailable(oID2),equalT o(res,vID),t3)	t3	t2	t3	UN	UN	
6	∃	**Happens**(re:UI: CarHire(oID3,vID),t4, ℜ(t2,t2+t_u))	t4	t3	t3 +1	UN	UN	

Note that, as a result of the update of the time variable of *Happens(rc:UI:CarRequest(oID1),t1,ℜ(t1,t1))* the monitor also sets the time boundaries LB and UB of the predicates *Initiates(rc:UI:CarRequest(oID1), equalTo(p,pID),t1)* and *Happens(ic:IS:FindAvailable(oID2,pID),t2,ℜ(t1,t2))* in the template. Then, when the event L10 is processed, the monitor sets the truth value of the predicate *Initiates(rc:UI: CarRequest(oID1),equalTo(p,pID),t1)* to *true* and updates its time stamp and source to 49 and RE, respectively. Subsequently, following the processing of events L11–L13, the above template for *B1* takes the following form:

Id		B1	T	F	DP			
u_t		{(oID1, op4), (pID, loc2), (vID, veh2), (oID2, op5), (p, p), (oID3, op6)						
P	**Q**	**SG**	**TS**	**LB**	**UB**	**V**	**SC**	
1	∀	**Happens**(rc:UI:CarReq uest(oID1),t1,ℜ(t1,t1))	49	49	49	True	RE	
2	∀	**Initiates**(rc:UI: CarRequest(oID1), equalTo(p,pID),t1)	49	49	49	True	RE	
3	∃	**Happens**(ic:IS:FindAv ailable(oID2,pID), t2, ℜ(t1,t2))	50	49	50	True	RE	
4	∃	**Happens**(ir:IS: FindAvailable(oID2),t3 , ℜ(t2,t3))	51	50	51	True	RE	
5	∃	**Initiates**(ir:IS: FindAvailable(oID2),e qualTo(res,vID),t3)	51	50	51	True	RE	
6	∃	**Happens**(re:UI: CarHire(oID3,vID),t4, ℜ(t2,t2+t_u))	t4	51	52	UN	UN	

At this point, the next event to be processed by the monitor is the event L14 in Figure 2. This event can be unified with the predicate *Happens(re:UI:CarHire(oID3,vID),t4,ℜ(t2,t2+1))* in the template and as it occurs before the upper time boundary of it (i.e., T=52), the truth value of the predicate is set to true, and its source and time stamp are set to RE and 52, respectively.

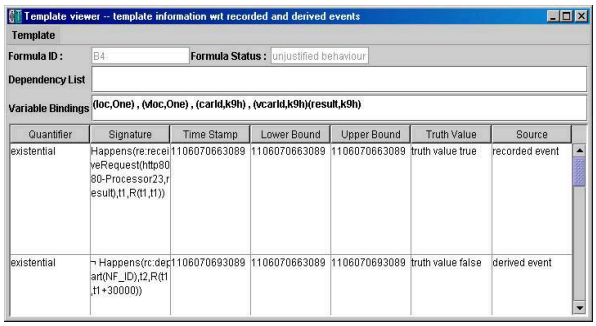

Figure 5. Deviation viewer

Following the establishment of the truth values of all the predicates in this template, the monitor can check it for possible violations. No violation, however, occurs until T=54, when the monitor derives the event *¬Initiates(ir:IS:FindAvailable(oID2), equalTo(res,veh2),51)* as we described in Section 2. Following the derivation of this event, the monitor will detect that it contradicts with the instantiation of the predicate *Initiates(ir:IS: FindAvailable(oID2),equalTo(res,vID),t3)* in the above template of *B1* and since this predicate is one of the conditions of *B1*, it will record a case of unjustified behaviour.

System providers may view templates with recorded deviations using the *deviation viewer* of the prototype that implements the monitoring framework. Figure 5 shows a snapshot of this component of the prototype.

5. Evaluation

To evaluate our monitoring framework we performed a series of experiments in which we used an implementation of the CRS example as a case study. In this case study, we extracted 7 behavioural properties from the BPEL specification of the composition process of CRS and specified 4 assumptions about the system and (these included the assumptions shown in Figure 1). The BPEL process of our case study and the behavioural properties and assumptions specified for it can be found at: www.soi.city.ac.uk/~am697/CRS_Case_Study.html.

The objective of our experiments was to measure: (a) the number of different types of violations that can be detected at run-time, (b) the average delay that occurs in detecting these violations, (c) the average delay for processing each event, and (d) the idle time of the monitor during the operation of the system.

In the experiments, we used a simulator that we developed to create sequences of events that can be generated by a given BPEL process. This simulator extracts all possible complete execution paths in a given BPEL process and expresses them as event calculus formulas. For each of the non-time variables in these formulas, the user must define the type (i.e., string, number or an enumeration) and size (i.e., the number of distinct elements) of its domain. For unconstrained time variables, the user must define the distribution function of

their values. For the formula *Happens(ic:p:A(id,x), t1, ℜ(t1, t1)) ∧ Happens(ic:p:B(id, y), t2, ℜ(t1, t1 + 10))*, for example, the user can declare the domains of *x* and *y* as strings with a maximum of 50 different values, and *t2* as a time variable uniformly distributed in the range *ℜ(t1, t1+10)*.

The simulator selects randomly a formula representing an execution path and generates all the events in it in the order they are expected. The time stamp of each of these events is computed randomly according to the distribution of the time variable of the relevant predicate. For predicates with unconstrained time variables (e.g. the predicate *Happens(ic:p:A(id,x), t1, ℜ(t1, t1))*) in the above formula, the simulator creates a random time stamp according to the distribution function of the time between different execution paths that is also set by the user. Finally, for non time variables, the simulator picks up randomly a value for each predicate variable from the respective variable domain.

Table 1. Basic time measures

Time	Meaning/Calculation
t_i^e	Time of event *i* as generated by the simulator.
T_s^m	Starting time of the monitor.
T_c^m	Current time of the monitor.
$t_i^{e(d)}$	Time of recording of an event *i* in the monitor's database. $t_i^{e(d)} = (t_i^e - t_0^e) + T_s^m$ where t_0^e is the first event that has been generated by the simulator.
t_i^M	Time when the monitor retrieves an event *i* from its database to process it.
T_S^{Fj}	Starting time of the decision procedure that checks for violations caused by a template *j* of a formula *F*
T_E^{Fj}	Time of completion of the check for violations caused by a template *j* for a formula *F*

Given the basic time measures shown in Table 1 that were taken in our experiments, we measured:

(i) The average delay in making a decision about possible violations in a template using the formula

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

where

- N is the number of the formula templates for which a decision has been made
- d_j is the delay in making the decision for template *j* that is computed as $d_i = \max(T_E^{Fj} - T_S^{Fj} + \max_{i \in Fj}(t_i^{e(d)}), T_E^{Fj})$ where *i* ranges over the events used to establish the truth values of the formulas in F_j).

(ii) The average delay in processing an event using the formula:

$$\text{e-delay} = \sum_{i=1,\ldots,K \text{ where } t_i^M - t_i^{e(d)} > 0} (t_i^M - t_i^{e(d)}) / K$$

where K is the total number of events.

(iii) The monitor's idle time using the formula

$$\text{idle-time} = \sum_{\text{event i where } t_i^{e(d)} - t_i^M > 0} (t_i^{e(d)} - t_i^M)$$

In the experiments, we ran 3 simulations of the CRS system. In the first of these simulations (*Sim 1*) we used a set of 50 customers, 20 cars and 3 car parks. In the second simulation (*Sim 2*), we increased the number of customers, cars and car parks to 100, 40 and 6, respectively. And in the third simulation (*Sim 3*), we increased these numbers to 200, 80 and 12, respectively. For each of these simulations, we generated 5 different sets of events having 1000-1020, 2000-2020, 3000-3020, 4000-4020, and 5000-5020 events respectively (small differences in the number of events of each simulation occurred due to the need to produce all the events of the last execution path selected by the simulator). The graphs in Figures 6-7 summarise our performance findings.

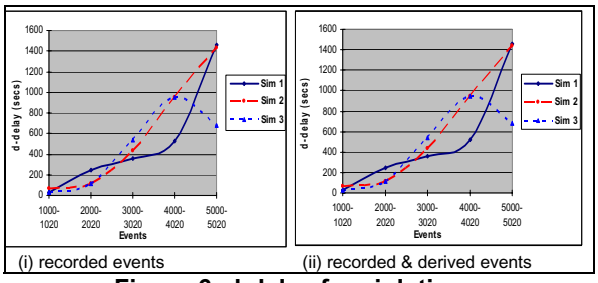

Figure 6. d-delay for violations

More specifically, Figure 6 shows the average delay for detecting different violations (*d-delay*) when only recorded events were taken into account (Figure 6.(i)) and when both recorded and derived events were taken into account (Figure 6.(ii)). The results were mixed. In *Sim 1*, we observed a linear increase in *d-delay* up to 4,000 events and then a steeper but linear increase for higher event numbers. In *Sim 2*, *d-delay* increased linearly with the number of the events all the way through and in *Sim 3*, we observed a drop in *d-delay* after 4,000 events. All simulations, however, demonstrated that the incorporation of derived events in monitoring did not affect d-delay. This was due to the fact that the derived event generator of the monitor runs in parallel with the monitoring process.

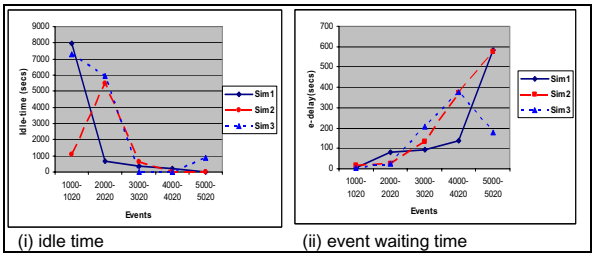

Figure 7. Monitor's idle and event waiting time

The aggregate monitor's idle time that is shown in Figure 7.(i) went down to very low levels for all three simulations after about 3,000 events and only in *Sim 3* it went up again to 1,000 secs in 5000 events. This finding was in line with the drop in d-delay that we observed in this simulation. The opposite trend was observed in *e-delay*: in *Sim 1* and *Sim 2* the average delay in processing an event was 3.2 and 17.7 secs at 1000 events and went up to 584 and 573.2 secs at 5000 events, respectively.

The number of different types of inconsistencies that were detected in our simulations is shown in Table 2. As shown in this table the number of violations of

assumptions due to recorded events (see columns T1 in Table 2) increased linearly with the number of events in all simulations. The number of violations by expected behaviour showed a less clear pattern as in two simulations (*Sim 1* and *Sim 2*) there was a drop in it after 4,000 events and only in *Sim 3* it increased linearly along with the number of events. Cases of unjustified behaviour were detected only in *Sim 2*.

Table 2. Number of violations

	Sim 1			Sim 2			Sim 3		
Events	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**	**T1**	**T2**	**T3**
1000-1020	8	0	0	9	0	0	7	0	0
2000-2020	13	13	0	13	54	2	15	38	0
3000-3020	18	47	0	25	39	38	25	62	0
4000-4020	26	107	0	28	100	0	25	96	0
5000-5020	40	83	0	40	74	0	27	123	0

The above results have been positive as they demonstrate that the size of the domains of the variables used in the formulas did not affect the performance of the monitor. Also there were cases where a drop in the delay of detecting violations was observed after a certain number of events (see *Sim 3*). Clearly, optimisations of the monitoring process (e.g. use of multiple parallel monitors) will be necessary to make our approach applicable to large scale systems, and further experiments will be required to confirm our initial results.

6. Related Work

Run-time requirements monitoring has been the focus of different strands of requirements engineering research. Most of the existing techniques (e.g. [5][6][11]) express requirements in the KAOS framework [7] as high level goals that must be achieved by a system. These goals are mapped onto events that must be monitored at run-time. Typically, the existing approaches assume that the events to be monitored are generated by special statements which must be inserted in the code of a system for this purpose (i.e., *instrumentation*) [6]. Note, however, that instrumentation cannot be always applied to SBS systems since typically service providers are not the owners of the services deployed by the system.

More recently, there has been research focusing on monitoring of SBS systems. Barezi et al [4], for example, have developed a monitoring tool that supports the run-time monitoring of assertions which are inserted into composition process of an SBS system specified in BPEL. When the execution of this process reaches the point where an assertion must be checked it calls an external service that checks the assertion. The execution of the composition process waits until the monitor returns the result of the check and it may continue or raise an exception depending on whether the assertion has been violated.

A different approach has been developed by Robinson [11]. In this approach, requirements are expressed in KAOS and analysed to identify *obstacles* for them (i.e., conditions under which they can be violated). If an obstacle is observable (i.e., it corresponds to a pattern of events that can be observed at run-time), it is assigned to an *agent* for monitoring. At run-time, an event adaptor translates web service requests and replies expressed as SOAP messages into events and a broadcaster forwards these events to the obstacle monitoring agents which are registered as event listeners to the broadcaster.

7. Conclusions

In this paper, we have presented the implementation and results of a preliminary experimental evaluation of a framework that we have developed to monitor requirements for service-based systems. This framework is applicable to service-based systems whose composition process is specified in BPEL and specifies the requirements to be monitored against such systems in event calculus. These requirements may be behavioural properties automatically extracted from the composition process of a service based system or assumptions specified by system providers. EC was chosen as the requirements representation language of our framework as it provides a formal temporal language amenable to reasoning based on the inference rules of first-order logic unlike other temporal logic languages.

We have implemented a prototype of this framework in Java™ and used it in a preliminary set of experiments to evaluate the performance of our framework. These experiments have confirmed that, due to its non-intrusive nature, our framework detects violations with some delay. On average, this delay has been found to increase linearly with the number of system events. Our experiments have also indicated no effect of the size of the domains of variables of formulas onto the delay of violation detection.

On-going work on the framework is concerned with: (i) the translation of service requirements expressed in the *WS-Policy* [3] and *WS-Agreement* [2] standards into the event calculus based language used by our framework, and (ii) the optimisation of the monitoring process deployed by the framework (e.g. introduction of formula template pruning capabilities).

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] Andrews T. et al. "Business Process Execution Language for Web Services", v1.1, http://www-106.ibm.com/developerworks/library/ws-bpel

[2] Andrieux A. et al. "Web Services Agreement Specification", Global Grid Forum, 2004,

http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf
[3] Bajaj S. et al. "Web Services Policy Framework", 2004, ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf
[4] Baresi L., Ghezzi C., and Guinea S. "Smart Monitors for Composed Services", *Proc. of 2nd Int. Conf. on Service Oriented Computing*, 2004
[5] Feather M.S., Fickas S., Van Lamsweerde A. and Ponsard C., "Reconciling System Requirements and Runtime Behaviour". *Proc. of 9th IWSSD*, 1998.
[6] Robinson W. "Monitoring Software Requirements using Instrumented Code". *Proc. of the Hawaii Int. Conf. on Systems Sciences*, 2002.
[7] Dardenne A., van Lamsweerde A. and Fickas S., "Goal-Directed Requirements Acquisition", *Science of Computer Programming*, 20:3-50, 1993.
[8] BPWS4J, http://alphaworks.ibm.com/tech/bpws4j
[9] http://logging.apache.org/log4j/docs/, September 2003
[10] Shanahan M. "The event calculus explained", *Artificial Intelligence Today*, 409-430, 1999
[11] Robinson W.N., "Monitoring Web Service Requirements", *Proc. of 12th Int. Conf. on Req. Engineering*, 2003
[12] Mahbub K., Spanoudakis G. "A Scheme for Requirements Monitoring of Web Service Based Systems", Technical Report, Computing Dept., City University, London, 2004.
[13] Mahbub K., Spanoudakis G. "A Framework for Requirements Monitoring of Service Based Systems" *Proc. of 2nd Int. Conf. on Service Oriented Computing*, 2004

Appendix: Event Processing Algorithm

```
EVENT_UPDATE(T,E): TL
/* E:event, T: formula template */
1.  TL = {} /* TL is a list of new templates created from T */
2.  for each predicate P in T such that P.V=UK do
3.    u_cur := imgu(E,P,u_t|P)
4.   if (P.tv is unconstrained) then
5.     if(u_cur ≠ ∅) then
6.       if partial(u_t|P) & u_cur – u_t|P ≠ ∅ then
7.      T' := T; TL := TL ∪ T'
8.   end if
9.       P.V:= ¬(P.NoQ xor E.NG); P.SC := RE;
10.       P.TS := E.TS; u_t := u_t ∪ u_cur;
11.       update boundaries of all other predicates Q in T
          where Q.tv is constrained by P.tv;
12.    end if
13.  else /* predicates with constrained time vars*/
14.   if (P.Q = ∃) then
        /* predicate type: ∃t.p(x,t) or ¬ ∃t.p(x,t) */
15.     if u_cur≠∅ & E not negated & (P.LB≤E.TS≤P.UB) then
16.        if partial(u_t|P) & u_cur – u_t|P ≠ ∅ then
17.           T' := T; TL := TL ∪ T'
18.          end if
19.        P.V := ¬ P.NoQ; P.SC := RE; P.TS := E.TS;
20.         u_t := u_t ∪ u_cur;
21.         update boundaries of all other predicates Q in T
            where Q.tv is constrained by P.tv;
22.      else
23.       if E.TS < P.UB then
24.          if   u_cur ≠ ∅ & E is negated & E.TS=P.TS+min_t then
25.             if partial(u_f|P) & u_cur – u_f|P ≠ ∅ then
26.                T' := T; TL := TL ∪ T'
27.               end if
28.             P.TS := E.TS; u_t := u_t ∪ u_cur;
29.          end if
30.        else
31.         if not  partial(u_t|P)  then
32.          P.V := P.NoQ;
33.          if P.TS= P.UB then P.SC := RE
34.              else P.SC := NF end if
35.          P.TS := P.UB;
36.           update boundaries of all other predicates Q in
              T where Q.tv is constrained by P.tv;
37.           end if
38.       end if
39.    end if
40.  else /* predicate type: ∀t.p(x,t) or ¬ ∀t.p(x,t) */
41.    if (u_t ≠ ∅) then
42.      if E.TS ≤ P.UB then
43.       if E.TS = P.TS+min_t then
44.         if u_cur ≠ ∅ then /* E is unifiable with p */
45.           if partial(u_t|P) & u_cur – u_t|P ≠ ∅ then
46.              T' := T; TL := TL ∪ T'
47.            end if
48.           if E not negated then
49.              P.TS := E.TS
50.           else
51.              P.V := P.NoQ; P.SC := RE; P.TS := E.TS;
52.              update boundaries of all other predicates
                 Q in T where Q.tv is constrained by P.tv;
53.           end if
54.           u_t := u_t ∪ u_cur;
55.         else /* E is not unifiable with p */
              if not partial(u_t|P) then
56.           P.V := P.NoQ; P.TS := P.TS +  min_t;
57.             P.SC := NF;
58.           update boundaries of all other predicates Q
                in T where Q.tv is constrained by P.tv;
59.          end if
60.        end if
61.       else /* E.TS > P.TS+min_t */
62.         if not partial(u_t|P) then
63.           P.V := P.NoQ; P.TS := P.TS +  min_t;
64.             P.SC := NF;
65.           update boundaries of all other predicates Q
                in T where Q.tv is constrained by P.tv;
66.          end if
67.       end if
68.     else /* E.TS >  P.UB */
69.      if P.TS = P.UB then
70.         P.V := ¬ P.NoQ; P.SC := RE;
71.          update boundaries of all other predicates Q in
             T where Q.tv is constrained by P.tv;
71.       else
72.         if not partial(u_t|P)  then
73.            P.V := P.NoQ; P.TS := E.TS – min_t;
74.            P.SC := NF;
75.              update boundaries of all other predicates Q
                 in T where Q.tv is constrained by P.tv;
76.          end if
77.       end if
78.      end if
79.    end if
80. end for
81. return TL
82. end EVENT_UPDATE
```

Symbols

imgu(e,p,u_t|P): Function that returns the most general partial unifier (i.e., a set of the form {(v_1, c_1), …, (v_k, c_k)}) of a predicate p in a template t with an event e that is compatible with the current unification of the variables of p in t (i.e. $u_{f|P}$).

u_t : u_t is a set {(v_1, c_1), …, (v_m, c_m)} representing the current bindings of the non time variables v_i in t (c_i is the value of v_i). In general, $m \leq n$ as there may be variables of t which have not been assigned any values at a given time.

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

partial($u_{t|P}$): Function that returns **True** if there is variable v of a predicate p for which there is no pair binding (v,c) in $u_{t|P}$ and **False** otherwise

P.NoQ: Variable indicating if the quantifier of P is negated (**True**) or not (**False**).

min_t: Minimum time unit in the clock of the SBS system

P.tv (Q.tv): time variable of predicate P (Q)