A declarative approach to timed-properties aware Web services composition

Ehtesham Zahoor, Olivier Perrin, Claude Godart

A declarative approach to timed-properties aware Web services composition

Ehtesham Zahoor, Olivier Perrin, and Claude Godart

LORIA, Nancy University, BP 239, 54506,
Vandoeuvre-lès-Nancy Cedex, France
{ehtesham.zahoor,olivier.perrin,claude.godart}@loria.fr

Abstract. In this paper we propose a paradigm shift for the timed-properties representation, computation and verification, introducing a declarative approach. The proposed approach provides a flexible event calculus based composition design, that allows for modeling different aspects such as the local temporal constraints for Web services with different synchronization modes and the global temporal constraints associated with the composition process. Further, given the composition design with timed properties representation, an event calculus reasoner can be used to compute a solution satisfying associated timed-properties. Then, in case of empty solution set the reasoner can provide near-miss models and/or unsatisfied clauses using a SAT solver to handle model verification. In addition, our approach also handles the run-time monitoring of the composition process and as our approach is declarative, it allows for recovery actions such as the re-computation of the composition plan to cater for detected run-time violations to design level service contracts.

Key words: Web services composition, Timed-Properties, Event Calculus

1 Introduction

Web services are in the mainstream of information technology and are paving way for inter and across organizational application integration. Individual services may need to be composed to satisfy user needs and as the Web services are autonomous, having local (including temporal) constraints and as the composition process may have some global constraints, the need to represent and compute an ordering satisfying the associated constraints is evident.

In this paper we propose a paradigm shift for the timed-properties representation, computation and verification, introducing a declarative approach. The proposed approach provides a flexible event calculus based composition design, that allows for modeling different aspects such as the local temporal constraints for Web services with different synchronization modes and the global temporal constraints associated with the composition process. Further, given the composition design with timed properties representation, an event calculus reasoner

can be used to compute a solution satisfying associated timed-properties. Then, in case of empty solution set the reasoner can provide near-miss models and/or unsatisfied clauses using a SAT solver to handle process model verification. In addition, our approach also handles the run-time monitoring of the composition process and as our approach is declarative, it allows for recovery actions such as the re-computation of the composition plan to cater for detected run-time violations to design level service contracts. Specifically we make the following contributions:

Composition design that can accommodate various aspects: The proposed approach provides a composition design that can accommodate various aspects including the representation of timed-properties for synchronous, *pull/push* based asynchronous and streaming Web services services, control and data based dependencies, data based constraints such as data expiry period, non-functional properties and other aspects. Further the proposed approach allows for specifying the recovery conditions and associated actions (called recovery constraints) to be used during the monitoring phase.

Declarative integrated approach: The proposed approach is declarative and uses the same set of constraints for the timed-properties representation, computation and verification within a unified framework. This allows for actions such as the re-computation of composition plan in case of run-time violations, which are difficult to handle using the traditional procedural approaches.

Event calculus for both composition and monitoring: Traditionally the web services composition problem is considered as a planning task, given a goal the planner can give a set of plans leading to the goal. Thus, in reference to our proposal, given a composition design with the representation of timed-properties using event calculus, "abduction reasoning" can be used to find a set of solutions. However, in case of timed-properties it may also be required to find out that if a given execution ordering is feasible or not, both in terms of conflicts and possible side-effects, and thus "deduction reasoning" on event calculus based model can be used. Further in case of detected violations during composition monitoring, "deduction reasoning" for side-effects calculation is needed.

SAT-solver based verification: The proposed verification approach relies on the SAT solvers to provide near-miss models for unsatisfied clauses. This allows not only for the conflicts (such as deadlocks) detection, but for identifying the hard constraints that should be relaxed to find a solution and for identifying other side-effects such as the data expiry.

Extensible approach: As we propose to use the event calculus, it allows for integrating the exisiting work on composition design [4], composition monitoring [8], authorization [1] and work on modeling other related aspects using the event calculus. Our proposed model thus can be modified and extended. Further, as the proposed verification approach relies on the SAT solvers, different SAT solvers can be added to handle the verification.

2 Motivation and related work

The motivation for our work stems from the process modeling, analysis and monitoring in a crisis situation and we present a emergency handling scenario that highlights the benefits of the approach. A crisis situation is, by nature, a dynamic situation especially in its first phases as we discussed in [14] and it requires the composition process to be more flexible to adapt to continuously evolving environment. Timed properties are one of the most important aspects in a crisis situation. As the information comes from different sources with different local temporal constraints, it is challenging to find a plan which respects the local constraints of different participating services and conforms to the process-level global temporal constraints, and to achieve that in a flexible way by not over constraining the composition process. In addition the successful execution of the identified plan is even more important, as in a crisis situation the services are more error prone to the response time delays, network failures and other unforeseen situations. Further due to the critical nature of these processes, ideally the composition process should be able to recover from and should provide recovery actions and alternatives to handle such situations.

A crisis scenario thus brings together the two related dimensions, as we discussed in [14]: *organization* and *situation*. The *Organization* dimension encompasses the design time composition modeling while the *situation* dimension require the composition process to measure and to adapt to continuously changing situation. This leads to the problem of Web services monitoring and the composition process should allows for a set of recovery actions to learn from and to cater for the run-time violations detected in the monitoring phase. The traditional workflow based approaches are procedural and thus are not flexible and some declarative approaches have been proposed [12, 10] and in [14] we discuss initial ideas for an event calculus based integrated declarative framework, called DISC, to bridge the gap between services composition and monitoring. In this work, we extend the DISC framework to handle timed-properties aware composition and propose that a declarative approach is more suitable to handle timed-properties aware composition especially in crisis scenario.

In the literature, there have been many proposed approaches to handle the timed-properties in services composition. The proposed approaches for the timed-properties based compatibility analysis of Web services composition include [5, 3, 11, 2] however as the authors criticized in [5] these approaches are limited and they assume the communicating services to be synchronous and authors proposed an asynchronous approach to the analysis of timed properties. Relatively few approaches tackle the efficient representation of timed properties in the composition process and traditional approaches assume the WS-BPEL process description for verification. In [6] authors introduced a formalism called WSTTS to capture timed behavior of Web services and then using this formalism for model-checking WS-BPEL processes, however we believe that the timed properties representation based on procedural approaches such as WS-BPEL are both rigid and it is very difficult to provide the recovery actions such as

re-planning (or re-computation of the composition plan) to handle the run-time violations by finding alternatives to satisfy the specified timed properties.

3 Motivating example

Let us consider a modified form of the emergency patient handling scenario we discussed in [13]. After a serious road accident, the patient is in critical condition and no documents other than his vehicle number are available to identify the patient. Patient is taken to nearby hospital in a remote region with limited resources and a composition process has been setup at the hospital to handle the emergency patient. In a typical Web services based setup, composition process interacts with different systems such as laboratory, vehicle registration, police department and others by the provided Web services.

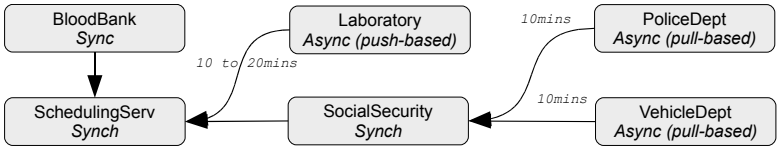

Fig. 1. Motivating example - emergency handling process

At the hospital, before the patient can be operated some blood tests are needed to identify any possible diseases, such as diabetes. Blood samples are thus sent to the laboratory department and the results can later be received using the Web service provided by the laboratory. Let us further consider that the laboratory provides a push-based asynchronous Web service, and results are provided (pushed) to the requester as soon as they are available normally between 10 to 20 minutes. Patient medical history can also be obtained by requesting the social security Web service, but for that the patient needs to be identified first. This can be done by contacting the vehicle and/or police department Web services to identify the owner information for the vehicle and to identify if the patient is indeed owner of the vehicle. These Web services are again asynchronous and they support pull-based asynchronous invocation and thus the results can be pulled 10 minutes after sending the request. The patient information is then communicated to social security Web service to get the medical history for the patient. Further, as the blood type of the patient is rare additional blood supply is requested by contacting the BloodBank Web service (see figure-1).

Once patient medical history or lab results are known, the scheduling service should be contacted for both scheduling the operation theatre and the surgery team. As the hospital has limited facilities and as there is another surgery already planned after 90 minutes, the composition process introduces some global temporal constraints, including:

- The surgery should start within next 25 minutes and it is suggested based on the nature of injuries to the patient that the surgery process should not last more than an hour.
- Once the surgery has been scheduled, it needs to be confirmed 5 minutes before the surgery begins.
- The schedule request can only be made if patient history/ lab results and additional blood supply information is available
- However, if there is some delay in obtaining the patient history/lab results, the delay can be notified to the scheduling service and the next surgery can either be rescheduled or transfered to some other surgery facility.

The emergency handling scenario presented above poses many challenges. First, specifying the exact sequence of activities to be followed as required by traditional procedural approaches, seems difficult as the solution depends on the specified constraints that are difficult to solve manually. For the declarative approach, these constraints mark the boundary of any acceptable solution and the reasoner can be used to provide a set of solutions satisfying associated constraints. Further, If there is no solution then either the constraints are too strict or there is some conflict in the specified model that needs to be identified. Then, any acceptable solution provided by the reasoner would be based on the design level service contracts and the actual service invocations can be different. Thus the process needs to be monitored to see if everything is working according to the plan and in case of any violations some recovery actions should be taken to cater for these violations.

4 DISC - Declarative Integrated Self-healing Composition

In this section, we will briefly discuss the DISC framework on which we will base our approach. The DISC framework, based on [14], is an event-oriented framework that aims to bridge the gap between the Web services composition and monitoring dimensions paving way for self-healing Web services composition. Proposed framework has three main stages, *Composition design*, *Instantiation and execution* and the *Composition monitoring*.

The composition process starts when the user provides the **composition design** backed up by an event calculus model, by specifying the basic entities (concrete Web service instances or abstract Web service types, called nodes) and associated constraints. These include the constraints to handle process choreography, for specifying non-functional properties, and the recovery constraints that specify the actions to be taken in case of monitoring violations.

The event calculus model for the composition design specified by the user can then be used to **instantiate and execute** the composition process. The instantiation phase involves both binding the nodes to the concrete Web service instances and for finding a solution to the composition process using the event

calculus reasoner. The Web services **composition monitoring** process phase monitors specified conditions and takes corresponding actions using the user preferences specified as recovery constraints.

5 Timed properties in the services composition

In this section we will extend the DISC framework to add the timed properties representation, computation, verification, and re-computation (as a form of recovery action). In reference to DISC framework different aspects of timed-properties are handled at different stages, for the representation of timed properties we will extend the composition design phase to add the event calculus models for handling timed properties. Then, the computation and verification of timed properties can be handled in the instantiation and execution phase and finally the re-computation can be added as a recovery action in case of run-time violations captured in the monitoring phase of the DISC framework, figure-2 details the proposed extensions to the DISC framework.

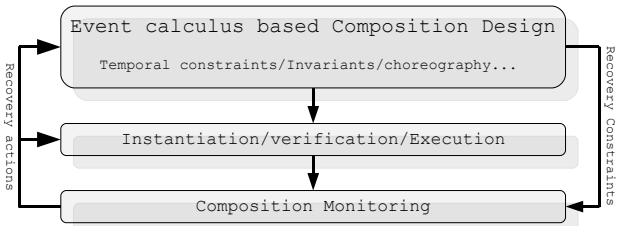

Fig. 2. Proposed framework stages

In order to detail our approach for the timed-properties aware composition, we will first discuss the composition design with focus on representing the timed-properties for various synchronization modes, such as push-based asynchronous services. The section-6 thus introduces the reader to the basic service composition model using event calculus and then extends it to include other aspects. We further introduce the concepts of implicit and explicit invariants to be used for verification. Once we have a timed properties representation using event calculus, the reasoner can be invoked to find a set of solutions and we discuss the computation and verification process in section section-7. Finally, in section-8 we discuss the re-computation of timed properties as a recovery action.

6 Event calculus based representation of timed-properties

In order to represent the timed properties of the Web services, we enrich the composition design in the DISC framework to add the temporal constraints and modeling various aspects such as asynchronous invocation of services.

6.1 Background and motivation

The proposed approach for the representation of timed-properties relies on the Event Calculus (EC) [7]. Event Calculus is a logic programming formalism for representing events and their side-effects and can infer "*what is true when*" given "*what happens when*" and "*what actions do*" (see figure 3). The "*what is true when*" part both represents the state of the world, called initial situation and the composition goal. The "*what actions do*" part states the effects of the actions. The "*what happens when*" part is a narrative of events.

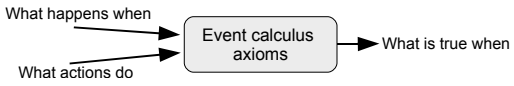

Fig. 3. Event calculus components

The EC is a first-order logic that comprises the following elements: $\mathcal{A}$ is the set of *events* (or actions), $\mathcal{F}$ is the set of fluents (fluents are *reified*), $\mathcal{T}$ is the set of time points, and $\mathcal{X}$ is a set of objects related to the particular context. In EC, events are the core concept that triggers changes to the world. A fluent is anything whose value is subject to change over time. EC uses predicates to specify actions and their effects. Basic event calculus predicates used for modeling the proposed framework are:

- $Initiates(e, f, t)$ - fluent f holds after timepoint t if event e happens at t.
- $Terminates(e, f, t)$ - fluent f does not hold after timepoint t if event e happens at t.
- $Happens(e, t)$ is true iff event e happens at timepoint t.
- $HoldsAt(f, t)$ is true iff fluent f holds at timepoint t.

Further, some event calculus axioms are available that relate the various predicates together. In reference to our proposal, the composition design serves as the initial knowledge ("*what is true when*" part) specifying the Web services being used in the composition process and dependencies that exist between them. The "*what actions do*" part specifies a set of actions (axioms) for handling the Web services invocation. Finally, the temporal ordering of the Web service invocations serve as the "*what happens when*" part.

The choice of EC is motivated by several reasons. First, EC integrates an explicit time structure (this is not the case in the situation calculus) independent of any sequence of events (possibly concurrent). Then, given the composition design (including timed-properties representation) specified in the EC, an event calculus reasoner can be used to instantiate the concrete composition. Further, EC is very interesting as the same logical representation can be used for verification at both design time (static analysis) and runtime (dynamic analysis and monitoring). Further, it allows for a number of reasoning tasks that can be

broadly categorized into deductive, abductive and indictive tasks. In the abductive reasoning, the "*what is true when*" and "*what actions do*" parts are provided and the event calculus reasoner can infer the narrative of events, i.e. "*what happens when*" which serve as the plan for the reasoning task while in the deductive reasoning task, "*what happens when*" and "*what actions do*" parts are provided and the event calculus reasoner can deduce "*what is true when*" part. In reference to our proposal, at composition design stage "abduction reasoning" can be used to find a set of plans and at the composition monitoring stage "deduction reasoning" can be used to calculate the effect of run-time violations.

6.2 Models

The event calculus models discussed in this paper are presented using the discrete event calculus language [9] and the reasoner used for instantiating the models is *DECReasoner*. In this work, we will only present the simplified models that represent the core aspects. We will intentionally leave out the event calculus supporting axioms, such as axiom to handle the case that once an action (such as service invocation) has been performed, it should not be performed again, if not explicitly specified. In the models, all the variables (such as service, time) are universally quantified and in case of existential quantification, it is represented with variable name within curly brackets, {variablename}. Further, for spacing issues we will abbreviate *Response* to *Resp*, and *Service* to *Serv* in some models.

Ground model: At a basic level, the composition process can be regarded as the invocation/reception of response from the participating Web services (for services with request-response invocation mode) and/or to invoke the services (with one-way invocation mode). For sake of simplicity, we will only consider in this paper the request-response invocation of Web services. The basic event calculus model to handle services invocation is as below:

Ground model - CM-1.0

```
sort service    fluent ResponseReceived(service)    event InvokeService(service)
Initiates(InvokeService(service),ResponseReceived(service),time).
```

The basic entities in the model are Web services, they can be regarded as a sort in the discrete event calculus language terminology. Then we define an event to specify the service invocation *InvokeService(service)*, a fluent *ResponseReceived(service)*, which specifies if we have received the response message from the Web service and an axiom which states that if the action *InvokeService(service)*, happens at some time then the fluent *ResponseReceived(service)* continues to hold after that time. Before going further, let us discuss how this basic model can be used for reasoning purposes by using the model below:

```
sort service    service Service1, Service2
event InvokeService(service)
Initiates(InvokeService(service),ResponseReceived(service),time).

!HoldsAt(RespReceived(Service1), 0)    !HoldsAt(RespReceived(Service2), 0) ;initial condition
!HoldsAt(ResponseReceived(Service1), 1) ;composition goal
```

In the model above, we add two instances of type service, called *Service1* and *Service2*, add initial condition that the fluent *ResponseReceived(service)* does not hold at time-point 0, a goal to the ground model above that the fluent must hold at time point 1 for services, and then invoke the reasoner. It gives us a plan, i.e. a temporal ordering, which shows that invoking the services concurrently at time-point 0, will result in receiving the response at time-point 1.

0
Happens(InvokeService(Service1), 0).
Happens(InvokeService(Service2), 0).
1
+ResponseReceived(Service1). +ResponseReceived(Service2).

The models presented above are synchronous; however in general the service invocation can be asynchronous and the composition process can request and later "pull" the data from provider or alternatively data is "pushed" to the process by service providers, when it is ready.

pull-based Asynchronous invocation: In order to model the pull-based asynchronous invocation, we update the model *CM-1.0*, and break down the invocation process by adding events and fluents for the sending request and then pulling the response. In the model below, we thus first introduce predicates that specify the synchronization mode for the Web service. Then we add another event to invoke asynchronous services and a new set of axioms to handle service invocation and then pulling for the response.

Asynchronous invocation (pull-based) - extends CM-1.0

fluent ResponseRequested(service)
event ReceiveResponsePull(service), InvokeAsynchService(service)
predicate IsSynchronous(service), IsASynchronousPull(service)

Initiates(InvokeAsynchService(service),ResponseRequested(service),time).
Initiates(ReceiveResponsePull(service),ResponseReceived(service),time).
Happens (ReceiveResponsePull(service), time1) → {time2} HoldsAt (ResponseRequested(service), time2) & time1 > time2.

Happens(InvokeService(service), time1) & Happens(ReceiveResponse(service), time2)→ time2 - time1 >= 20.

The second last axiom specified in the above model specifies that the event *ReceiveResponsePull(service)* can only happen if we have already requested for the response. Other axioms specify the response request and the eventual "pull" for the response message. The last axiom models the minimum time after which the response data is available to be pulled.

push-based Asynchronous invocation: In order to model the push-based asynchronous invocation, we introduce the concepts of (bounded) queues that can be used to store the pushed data from the service providers and composition process can then use the data from the queues. We use the pull-based asynchronous model and add the fluent and corresponding event to model the

queues and data being pushed to queues by the service providers. In the updated model, the process first sends the request, *InvokeAsynchService(service)*, and then the response is pushed to the process queue, *PushResponse(service)*, between the specified time intervals. Once the data is available in the queues, *HoldsAt(ResponsePushed(service), time2)*, the response can then be retrieved from the process queue, *ReceiveResponsePush(service)*.

Asynchronous invocation (push-based) - extends CM-1.0

fluent ResponsePushed(service)
event PushResponse(service), ReceiveResponsePush(service)
Initiates(PushResponse(service), ResponsePushed(service),time).
Initiates(ReceiveResponsePush(service),RespReceived(service),time).

Happens(ReceiveResponsePush(service), time1) $\rightarrow$ {*time2*} *HoldsAt(ResponsePushed(service), time2)* & *time1* > *time2.*

Modeling other aspects: Using event calculus also allows for various related aspects, below we will briefly discuss the modeling for only some core aspects due to space limitations. For **data modeling**, event calculus can be used to represent the *data based dependencies, data expiry, data values* and others. In order to model data values, we can define two new sorts, *request* and *response*, change the service invocation axioms to cater for them and introduce new instances of these sorts that specify the request and response data values. In order to model the data expiry, we can introduce the *fluent ResponseValid(service)*, and add an event that invalidates the data after a certain time. Further, in order to model the **streaming services**, we can add the data validity axioms to asynchronous Web services modeling and add an axiom to re-invoke the service once the data expires. For a detailed discussion about how different **control sequences** such as *loops* and others can be modeled using event calculus, see [4].

Then, we can also add **invariants** to the composition model that specify the conditions that should be respected at a particular time-point or for the complete process. Invariants can be explicitly added or they can be implicitly inferred by the reasoner and the invariants are respected while the reasoner attempts to find a solution. The first axiom below serves as an invariant and specifies that the service S1 has dependency on service S2.

Invariants/recovery constraints specification

Happens(InvokeSynchService (S1), time1) $\rightarrow$ {*time2*} *HoldsAt(ResponseReceived (S2), time2)* & *time1* >= *time2.*
Happens(StartInvocation(S1), time1) & *!HoldsAt(ResponseReceived(S1), time2)* & *time2 -time1 = 10* $\rightarrow$ *Happens(Terminate(), time2).*

The second axiom in the above model is a **recovery constraint** that specifies the condition to monitor and the corresponding recovery action to be taken during composition monitoring phase. Recovery constraints take the form of axioms with an *activation condition* part and an *action* part. As an example, the second axiom above is a recovery constraint that specifies to *terminate* the execution in the case of response time delay for service S1. The axiom comprises an

activation condition and the corresponding action and can be regarded as *activation condition → action*. The action in above axiom, *Happens (Terminate(), time2)*, specifies to terminate the execution if there is delay in expiry time. There are other actions possible, such as *ignore*, *retry* and others. In this paper, we will only consider the *recompute (replan)* action.

6.3 Example

Let us now review the motivating example and discuss the event calculus model with timed-properties representation. In order to keep the model short and simple we won't introduce the modeling of response data and consider the multiple invocations of *SchedulingService* with different parameters, as multiple service invocations. In the model below we first define the instances of the sort *service* that specify the Web service instances, synchronization modes and the local temporal constraints associated with each service:

Motivating example - participating entities

service SocialSecurity, VehicleDept, ReSchedulingServ ...
IsSynchronous(SocialSecurity), IsPullBasedASynchronous(VehicleDept), ...
Happens(InvokeAsynchService(VehicleDept), time1) & Happens (ReceiveResponsePull(VehicleDept), time2)→ time2 - time1 = 10...

Next, we introduce the dependencies between different services (considered as *Invariants*), *SocialSecurity* service has dependency on either *Police* or *VehicleDept* service, while the *SchedulingService* has dependency on either *SocialSecurity* or *Laboratory* services. We can model the dependencies between *SchedulingService* and scheduling confirmation service in a similar fashion, space limitations restrict us to discuss them further.

Motivating example - dependencies/invariants

Happens(InvokeSynchService (SocialSecurity), time1) → {time2} (HoldsAt(ResponseReceived (VehicleDept), time2) | HoldsAt(ResponseReceived (PoliceDept), time2)) & time1 >= time2.

Happens(InvokeSynchService(SchedulingServ),time1) → {time2} (HoldsAt(ResponseReceived (SocialSecurity), time2) | HoldsAt(ResponseReceived (Laboratory), time2)) & time1 >= time2.
...

Then, we add the initial situation for the fluents, that they does not hold at time point 0, and specify that the invocation of bloodbank, laboratory and police/vehicle department starts at time point 0. Further, as the surgery must start in 25 minutes, so we need to either confirm/reschedule other surgery at time-point 20, we thus specify a goal for the composition process that the response from either of these services is available at time-point 20.

Motivating Example - Initial situation and composition goal

!HoldsAt(ResponseRequested(service),0). !HoldsAt(ResponseReceived(service),0)...
Happens(InvokeAsynchService(PoliceDept), 0), Happens(InvokeAsynchServ(Laboratory), 0)...
HoldsAt(RespReceived(SchedulingConServ),20) | HoldsAt(RespReceived(ReSchedulingServ),20).

We can further add a recovery constraint to recompute the solution if the response from both police and vehicle department services is not available at time point 11. This constraint will be used during the Web services composition monitoring phase.

7 Computation and Verification

The event calculus model used for the timed properties representation can then be used for the computation and verification of timed-properties aware services composition. The computation phase is similar to the instantiation phase for the DISC framework; the event calculus reasoner is used to find a set of acceptable solutions to the composition process and a particular solution is chosen for execution. However, if there are some conflicts in the representation and/or the specified constraints are too strict, this leads to empty solution set and requires the verification of the composition representation to identify any conflicts or hard constraints.

Our approach to verification relies on the SAT solvers to provide a set of near-miss models and/or unsatisfied clauses. We introduced the concept of Invariants that are axioms that should be respected at a particular time-point or for the complete process. Invariants are respected by the reasoner for providing a solution, however if no solution is found and/or if the partial process plan is already specified, the reasoner can generate near-miss models highlighting the constraint not being satisfied. As an example consider the temporal constraint added to the composition process that the services, S1 and S2 should not execute concurrently. In case of planning, the reasoner will only generate the solutions that will respect this constraint however, if no such solution exists and the only solution is to execute them concurrently to achieve the specified goal, the planner can return a near-miss model highlighting the strict constraint.

Delegation of verification task to the SAT solver has many benefits. First, in relation to the proposed implementation framework, the *DECReasoner* attempts to find a solution by transforming the problem into a SAT problem and invoking SAT solver for the solution, thus the same SAT encoding can be used for verification purposes. Then, it provides an highly extensible approach, same SAT encoding can be either analyzed by multiple solvers. Further, it allows not only for the conflicts (such as deadlocks) detection, but allows for identifying the hard constraints that should be relaxed to find a solution and for identifying other side-effects such as the data expiry and others.

7.1 Example

Invoking the reasoner for the event calculus model for the motivating example gives us a set of models including the following:

```
0
Happens(InvokeAsynchService(Laboratory), 0). Happens(InvokeAsynchService(PoliceDept), 0).
Happens(InvokeAsynchService(VehicleDept), 0). Happens(InvokeSynchService(BloodBank), 0).
1
+ResponseReceived(BloodBank). +ResponseRequested(Laboratory). ...
2
3...
10
Happens(ReceiveResponsePull(PoliceDept), 10).
Happens(ReceiveResponsePull(VehicleDept), 10).
11
+ResponseReceived(PoliceDept).
+ResponseReceived(VehicleDept).
12
Happens(InvokeSynchService(SocialSecurity), 12).
13
+ResponseReceived(SocialSecurity).
Happens(InvokeSynchService(SchedulingServ), 13).
14
+ResponseReceived(SchedulingServ).
15
16
17
Happens(InvokeSynchService(SchedulingConServ), 17).
18
+ResponseReceived(SchedulingConServ).
19
20
+ResponseReceived(Laboratory).

3 predicates, 0 functions, 3 fluents, 5 events, 46 axioms
encoding 2.9s solution 0.8s total 5.7s
```

The model above shows that there exists a solution in which the response can be received from Police/Vehicle department Web service for identifying the patient and thus retrieving the patient history from social security Web service and in turn scheduling the patient. The Laboratory Web service can take 10-20 minutes and in the worst case scenario the results are obtained after 20 minutes, so it is not chosen as a possible solution.

Before going further, let us also discuss how the verification of event calculus model can be done using the reasoner for finding the near-miss models. We introduce a cyclic dependency leading to deadlock in the event calculus model for the motivating example by specifying that the BloodBank service has dependency on Scheduling service and vice-versa. Then event calculus reasoner returns the a near miss model (using the Walksat solver) with following information:

```
2 unsatisfied clauses:
-1830 0: (!ReleasedAt(ResponsePushed(Laboratory), 0)).
1803 -4 0: (HoldsAt(ResponseReceived(SchedulingServ), 0) | !Happens(InvokeSynch er-
vice(BloodBank), 0)).
```

The information returned by the reasoner along with near-miss model shows that there are two clauses that are causing the reasoner to fail. In the discrete event calculus terminology, all the fluents are, by default, subject to common-sense law of inertia and the release from the law should be explicit. Once released, the fluent can have either truth or false value. So the first clause specifies that if the fluent, *ResponsePushed(Laboratory), 0)*, can be released at time-point 0 (and thus can hold at time-point 0) it would allow us to receive response and thus invoke the bloodbank service. The second related clause, specifies that either

ResponseReceived(SchedulingServ) does hold at time-point 0 or the bloodbank service is not invoked at timepoint 0, *!Happens(InvokeSynch Service(BloodBank), 0)*. The reasoner is thus trying to break the deadlock by specifying one of two cyclic dependencies must be removed giving much information for the resolving the conflict. The SAT solver we have used is Walksat and as discussed before some other more efficient solver can also be integrated to give event better verification feedback.

8 Re-computation

Timed-properties aware representation, computation and verification that we have discussed so far is based on design level service contracts and they may change at the execution time for the actual service invocations. An important aspect of our approach is to thus handle the run-time monitoring of the composition process and to provide a set of recovery actions to cater for run-time violations. The Web services **composition monitoring** phase is divided into three phases: *Detection* phase is responsible for detecting the monitoring violation by comparing the event logs with the initially instantiated plan, the *Side-effects calculation* phase is responsible to deduce the side-effects of the detected monitoring violation. In reference to our proposal, an important application of side-effects calculation monitoring sub-phase is the model verification due to updated situation. Finally the *Recovery* stage is responsible for using the user preferences for recovery action, specified as recovery constraints, to cater for and recover from the violation. Space limitations restrict us to detail these phase further and in reference to our proposal, we will only consider the case when the associated recovery action is re-computation.

8.1 Example

In reference to the motivating example instantiated plan (see section 7), let us consider that after 10 minutes, the patient history is not available as the services are not respecting design time service agreements and thus at monitoring phase a violation is detected at time-point 11. To handle this violation the action specified by the user is to recompute the plan and reasoner is invoked again after adding the updated information to the plan. There are two cases, as the laboratory service takes 10-20 minutes the result can be obtained after 10 minutes. So if we consider this case the first re-computed plan shows that we can still invoke *SchedulingConServ* respecting constraints. However if data is not available from the laboratory service as well, the second re-computed plan below shows the invocation of *ReSchedulingServ* :

... Happens(PushResponse(Laboratory), 10). +ResponsePushed (Laboratory). ... Happens (InvokeSynchService(SchedulingConServ), 19). +ResponseReceived(SchedulingConServ).
... Happens(InvokeSynchService(ReSchedulingServ), 19). +ResponseReceived(ReSchedulingServ).

9 Conclusion

In this paper we propose a declarative approach for the timed-properties representation, computation and verification. The proposed approach provides a flexible composition design, based on event calculus, that is able to represent various aspects including the local temporal constraints for Web services with different synchronization modes such as synchronous, asynchronous and streaming Web services and the global temporal constraints associated with the composition process. Then given the composition process representation, an event calculus reasoner can then be used to compute a solution satisfying associated timed-properties and to verify the event calculus model. The proposed verification approach relies on the SAT solvers to return near-miss models and/or unsatisfied clauses to identify any conflicts in the model. Further, our approach also handles the run-time monitoring of the composition process and in case of run-time violations, temporal constraints can be updated (if possible) and a solution can be re-computed.

References

1. A. K. Bandara, E. C. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:26, 2003.
2. B. Benatallah, F. Casati, J. Ponge, and F. Toumani. On temporal abstractions of web service protocols. In *CAiSE Short Paper Proceedings*, 2005.
3. L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are two web services compatible? In *TES*, pages 15–28, 2004.
4. N. K. Cicekli and Y. Yildirim. Formalizing workflows using the event calculus. In *DEXA*, 2000.
5. N. Guermouche and C. Godart. Asynchronous timed web service-aware choreography analysis. In *CAiSE*, pages 364–378, 2009.
6. R. Kazhamiakin, P. K. Pandya, and M. Pistore. Representation, verification, and computation of timed properties in web. In *ICWS*, pages 497–504, 2006.
7. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.
8. K. Mahbub and G. Spanoudakis. A framework for requirents monitoring of service based systems. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 84–93, New York, NY, USA, 2004. ACM.
9. E. T. Mueller. *Commonsense Reasoning*. Morgan Kaufmann Publishers Inc., 2006.
10. M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops*, 2006.
11. J. Ponge, B. Benatallah, F. Casati, and F. Toumani. Fine-grained compatibility and replaceability analysis of timed web service protocols. In *ER*, 2007.
12. W. M. P. van der Aalst and M. Pesic. Decserflow: Towards a truly declarative service flow language. In *The Role of Business Processes in Service Oriented Architectures*, 2006.
13. E. Zahoor, O. Perrin, and C. Godart. Mashup model and verification using mashup processing network. In *CollaborateCom2008*. ACM, 2008.
14. E. Zahoor, O. Perrin, and C. Godart. An integrated declarative approach to web services composition and monitoring. In *WISE*, pages 247–260, 2009.