An event-based reasoning approach to Web services monitoring

Ehtesham Zahoor, Olivier Perrin and Claude Godart
Université de Lorraine, Nancy 2, LORIA
BP 239 54506 Vandoeuvre-lès-Nancy Cedex, France

{*ehtesham.zahoor, olivier.perrin, claude.godart*}*@loria.fr*

***Abstract*—In this paper, we propose an event-based framework that allows to specify and reason about the monitoring properties during composition process execution. The proposed approach is highly expressive and allows to specify monitoring properties that can be based on either functional or non-functional requirements, allows multi-level detection of any violation, allows to calculate effects of any such violation on the overall process execution and to recover from it using a set of recovery actions. The choice of a reasoning based approach allows to foresee the effects of violations and respects any functional and non-functional constraints associated with the process, when performing recovery. In addition, as the approach builds upon an event-based declarative framework called DISC, it results in an integrated approach as both composition design and monitoring framework are event-based.**

I. 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 form new added value processes and the need to monitor the Web services composition process during execution stems from two major objectives. At one hand continuously monitoring the resource utilization, SLA's violation, or some domain specific Key Performance Indicators (KPI's) may be required to measure the performance or to fulfill some domain specific monitoring requirements.

Then, as the Web services are autonomous and only expose their interfaces, composition process is based on design level service contracts and the actual execution of composition process may result in the violation of the design-level services contracts due to errors such as network or service failures, change in implementation or other unforeseen situation. This highlights the need to detect the errors and react accordingly to cater for them. The reaction may include to calculate the effect the violation has on the overall process execution and then to recover from it. Traditional approaches for the composition monitoring are proposed as an extension to some particular run-time and are tightly coupled and limited to it. In contrast the use of an event-based approach works on the message-level and thus is unobtrusive, independent of run-time and allows for integration of other systems and processes, as discussed in [1]. Then, the traditional monitoring approaches build upon composition frameworks that are highly procedural, such as BPEL, an this in-turn poses two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not first-class objects of the composition design framework (but rather defined and extracted from it) and functional and non-functional properties are not expressed in terms of events and their effects. Then, the use of procedural approach for process specification make it very difficult to use recovery actions such as re-planning or alternate path finding as we discussed in [2]. In this paper, we propose an event-based framework that allows to reason about the monitoring properties during execution. The proposed approach builds upon an event-based declarative framework, called **DISC**, and as both composition specification and monitoring framework are events based, this results in an integrated approach that is easily able to monitor, detect, calculate side effects and recover from the monitored violations. Specifically we make the following contributions that highlight the uniqueness of our approach:

Event-based monitoring: Event-based approach results in message-level monitoring approach that allows integration of other systems. In addition, the proposed monitoring approach is built upon an event-based declarative composition design and this results in an integrated approach that allows to reason about the events and does not require to define and extract events from process specification, as the events are first class objects of both design and monitoring frameworks.
Properties specification: The proposed approach is highly expressive and allows the specification of monitoring properties that are based on both functional and non-functional (such as temporal, security or their combinations) requirements. These properties are expressed as event-calculus axioms and can be added to the process specification both during process design and during the process execution.
Measurement and anomalies detection: The proposed approach both allows for KPI's measurement[1] (that may be needed for process evaluation or result in proactive detection

[1]We will collectively use the term KPI's measurement in the paper, however it can signify monitoring the resource utilization, SLA's fulfillment, or some domain specific KPIs that may be required to measure the performance or to fulfill some domain specific monitoring requirements.

of any violations) and the detection of violations once they happen. Different levels of detection are provided such as detection to the process execution plan, detection to the violations based-on any properties and events added during process execution and others.

Violation side-effects: As the proposed approach uses reasoning at the execution-time, it allows for calculating the effects any monitored violation has on the process execution. This also allows to cater for the "ripple effect" any violation has on the process execution, and for proactive detection of any possible violation that is bound to happen later in the process execution, as a result of current detected violation.

Forward and backward recovery schemes: Once a violation is detected and a recovery solution (re-planning or alternative path finding) is sought, the proposed approach allows both to find a new solution based on the current process state (thus specifying what steps should be taken now to recover from the violation and hence termed forward recovery) or to backtrack to some previous activity (if possible) and try to find a new from there. Then, any recovery solution takes care of the process specification and the associated QoS properties, when performing recovery.

Implementation architecture: The proposed approach uses the DECReasoner as the event-calculus reasoner, however as we discussed in [2], [3] the event-calculus to SAT encoding process provided by the reasoner, does not scale well. In this work, we have modified the DECReasoner code to gain substantial performance improvement as evident in performance evaluation results (Section-IX). Further, we have presented a real world crisis management case-study and discussed how a process-based approach can be beneficial.

II. Motivating example

The motivation for our work originates from the need for the process modeling, analysis and monitoring in a crisis situation. A crisis situation is highly *dynamic* and it demands for a process that is possibly partially defined, is characterized by temporal and security constraints and uncertainty, multiple and possibly changing goals, and thus requires the composition process to be more flexible to adapt to a continuously evolving environment. The crisis scenario brings together two related dimensions of *organization* and *situation measurement*. The *organization* dimension encompasses the design-time composition process modeling while the *situation measurement* requires the composition process to measure and adapt to continuously changing situation. In our previous work, we have proposed an event-based declarative framework, called **DISC**, for self-healing Web services composition [2], and later extended the framework to handle security and temporal requirements in the services composition [3]. While we discussed some initial ideas for an event-based monitoring but the focus of our previous work was to justify the usage, expressiveness and benefits of an integrated and declarative approach and in this paper, we propose and discuss the event-based monitoring framework.

For the motivating example, we consider a composition process being setup to semi-automate the recovery plan for the Australian National Herbarium (ANH), Canberra[2]. The possible threats to the ANH collection include the bushfire and lighting strikes due to its location at the base of Black Mountain and close proximity to bushland. The ANH recovery plan is in the document form and while we stress the need for a process based approach to the crisis handling, space limitations restrict us to go that far and we will only consider a simple case-study for the recovery of the items for priority salvage and treatment at ANH, after an unfortunate fire accident. These items include the *Type specimens* and as a result of fire, they may get smoke-damaged and the recovery plan suggest to either freeze or transfer them to some other location.

Once the fire-alarms are activated, the composition process contacts the Web services provided by different services such as fire-brigade, police, ambulance and also invokes a service to call emergency handling staff. Then, once the fire-brigade has reached the site and after partial or complete fire containment and given clearance to enter the building, the collection recovery coordinator (CRR), facilities coordinator (FC), Salvage Controller (SC) enter the premises. The role of CRR include the overall management of recovery process while the FC is responsible for ensuring that the recovery teams have proper equipment for the recovery. Then, the SC is responsible for initial cleanup, salvaging, sorting and stabilization of materials by creating recovery teams. The recovery plan suggests the SC can be either Librarian or Collection Manager at ANH. In this work we would not address how such composition can be modeled in a declarative and flexible way [2], but the focus is to highlight the need of monitoring certain properties during execution as crisis handling process can be highly dynamic. Due to importance and priority of specimens and actively deteriorating situation, we consider the following monitoring constraints to be imposed on the composition process:

- REQ-1: The smoke damaged, wet specimens should be treated as soon as possible and any delay of more than an hour may result in serious damage to the specimens.
- REQ-2: Once the response message (containing updated arrival information) from the fire-brigade and emergency contacts calling services have been received the constraints mentioned above are re-evaluated.
- REQ-3: These constraints must be respected for any recovery solution to cater for monitored violations.

For this scenario, these constraints are already known and can be defined at the design-time using a event-based approach [2]. However, the successful execution of the process is challenging provided highly dynamic and con-

[2]http://www.anbg.gov.au/cpbr/disaster-plan/

tinuously changing situation. For instance, the time-taken by the fire-brigade and emergency staff to reach the site can be somewhat defined and estimated but the time-taken for the fire-containment process itself is highly relative. Then, it may not be possible to define all the monitoring properties during composition design and even if the design-level constraints are respected, the occurrence of external events during process execution can have impacts on the process execution. We will use this example as base for describing different aspects of the proposed framework.

III. Related work

The problem to effectively monitor and recover from the anomalies during process execution is highly active and widely studied research direction and in a broader sense it spans different related domains. Workflow or Process management systems, in general, rely on an exception handling approach to specify exceptions and handles during process design [4], [5], [6], [7]. Further, for the self-healing systems a number of approaches have been proposed for monitoring, diagnosis and recovery from errors during execution [8], [9]. A detailed discussion about monitoring and recovery in different systems can be found in [10] and space limitations restrict us to the discussion of monitoring and recovery mechanisms for only services based processes, where different services are composed using the approaches such as BPEL. Traditional services composition approaches, such as BPEL, build upon the exception handling approach for errors handling and allow to define different type of exception handlers and corresponding actions based on process state. However, it may not be always possible to foresee errors and specify the exceptions at the design-time. In the literature, a number of composition process monitoring approaches have been proposed, but in general, they are proposed as an extension to some particular run-time and are tightly coupled and limited to it [11], [12], [13], [14]. As a result and they do not consider other sub-systems and or processes that can be used for monitoring [1].

Then, the traditional monitoring approaches [15], [16], [17] build upon composition frameworks that are highly procedural, such as BPEL, an this in-turn poses two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not part of the composition framework and functional and non-functional properties are not expressed in terms of events and their effects. Secondly the use of procedural approach for process specification does not bridge the gap between organization and situation in a way that it is very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time, and it does not allow for a reasoning approach allowing for effects calculation and recovery actions such as re-planning or alternate path finding as we discussed in [2]. In [1] authors proposed an event based monitoring approach that works on the message-level and thus is unobtrusive, independent of run-time and that highlights the need and motivation for using an event-based monitoring framework. However the approach aims to extract and define events from procedural process specification, while our approach builds on an event-based framework and events are first class objects in both composition design and monitoring framework. This allows to reason about events during execution and allows for effects calculation and different types of recovery actions. In [18] authors attempt to add monitoring directives to a declarative approach but still the approach lacks expressiveness and does not allow for recovery actions. Our work can be compared to PAWS framework [19], in which authors propose to add annotations to the BPEL process to handle services replacement in case of run-time failure. However, as the approach is based on BPEL and is procedural, it allows for limited recovery options (such as service replacement and not re-planning or alternative path finding) and effects calculation once a violation is detected. In [10] authors proposed a model based approach for repair by exploiting information about the causes of process and deriving repair strategies based on process structure, however the approach builds upon BPEL and PAWS and thus does not allow to reason about monitoring properties and allowing for effects calculation and different recovery schemes.

IV. Background

A. The DISC and DISC-SeT frameworks

We have proposed a declarative event-oriented framework, called **DISC** (*Declarative Integrated Self-healing web services Composition*) [2], that serves as a unified framework to bridge the gap between the process design, verification and monitoring and thus allowing for self-healing Web services composition. The proposed **DISC** framework has four main stages, *Composition design*, *Instantiation and Verification*, *Execution* and *Composition monitoring* (see Figure-1).

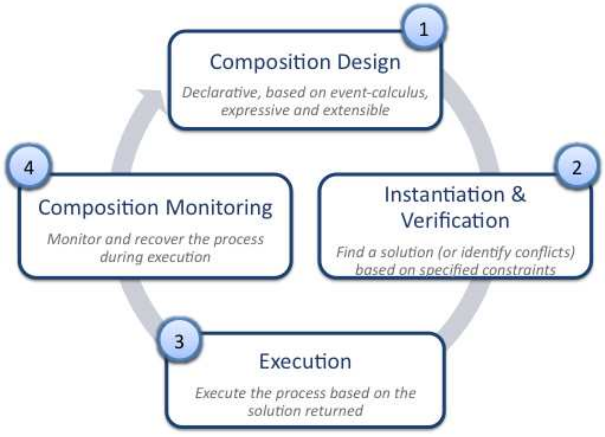

Figure 1. The DISC Framework

The composition process starts when the user provides the *composition design*, which allows to accommodate various aspects such as partial or complete process choreography and exceptions, data relationships and constraints, Web services dynamic binding, compliance regulations or other non-functional aspects (see Figure 1-1). The process design (and so as other process life-cycle stages) is based on event calculus and a detailed discussion about the motivation for the usage of event-calculus and models for different aspects can be found in [2]. The event-calculus model for the composition design is then used to instantiate, verify and execute the composition process (see Figure 1-2). The instantiation phase involves finding a solution to the composition process using the event calculus reasoner. As our proposal allows for specifying only the boundaries to the composition process and, the instantiation may result in a number of solutions and the user can be given option to choose one solution, this choice can also be based on some other criteria such as overall cost, minimal time and others. The chosen plan is then executed and the plan generated by the reasoner serves as a script for the execution of the process (see Figure 1-3). However, If there are some conflicts in the composition design and/or the specified constraints are too strict, this leads to empty solution set and the the reasoner can return near-miss models and/or conflicting clauses highlighting the strict constraints or the conflicts. The process is then monitored during execution using an event-based monitoring approach, that is the focus of this paper (see Figure 1-4) and in case of any violation, process specification can be updated and re-instantiated to find an updated solution. We have also extended the **DISC** framework to propose the **DISC-SET** framework [3] to add the security and temporal properties representation, computation and verification.

B. Event-calculus

The **DISC** framework is based on event-calculus (EC) which is a logic programming formalism for representing events and their side-effects. It comprises the following elements: $\mathcal{A}$ is the set of *events* (or actions), $\mathcal{F}$ is the set of fluents, $\mathcal{T}$ 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. A detailed discussion about event-calculus can be found in [18] and we discussed the motivation for usage of event-calculus in [2]. Some basic event calculus predicates used for modeling the proposed framework are:

- $Initiates(e, f, t)$ - fluent f holds after time point t if event e happens at t.
- $Happens(e, t)$ specifies that event e happens at time point t.
- $HoldsAt(f, t)$ is true iff fluent f holds at time point t.

Given an event-calculus based specification, different reasoning modes can be used and Figure-2 summarizes different reasoning modes used during different process stages.

Phase	Reasoning approach
Solution computation / design-time verification	Abductive resoning to find plan for the specified goal
Process change during execution	Abductive reasoning to identify conflicts and to check if goal is still acheivable
Effects calculuation	Deductive reasoning by adding partial plan and violation
Recovery (replanning)	Abductive Reasoning to find a new plan based on updated process state

Figure 2. EC Reasoning modes for the DISC framework

C. Example

Using the **DISC** framework for the process specification we can model the composition process for the motivating example in a declarative and flexible way [2]. We can define events/fluents for services invocation, for activities representation and domain specific events such as modeling the arrival of support staff. For simplicity, we will abstract different processes such as fire-containment, recovery and the responsibilities of the collections coordinator and other users as activities however they can be service based sub-processes. We consider the following design-level constraints; the firefighters arrive within 15 minutes after the service invocation, the support staff arrival varies and may take 25 minutes for their arrival. Then, the time taken for the fire-containment process is highly relative and provided the central location of priority salvage items, it may be possible to contain the fire within 20 minutes. Finally, the recovery process may take 10 minutes once fire is (possibly partially) contained. Space limitations restrict us to discuss the design model in detail and we here present the instantiated solution returned by the reasoner, the serves as a plan for process execution and we will use it as a base for monitoring violations detection. The instantiated model is shown below:

0 *Happens(InvokeService(CallEmergencyStaff), 0)*
Happens(InvokeService(FireBrigade), 0).
1 *+ResponseReceived(CallEmergencyStaff). +ResponseReceived(FireBrigade).*
Happens(ValidateAndUpdatePlan(CallEmergencyStaff), 1).
Happens(ValidateAndUpdatePlan(FireBrigade), 1).
2 *+PlanValidatedAndUpdated(CallEmergencyStaff)...*
...
15 *Happens(Arrives(FireFighters), 15).*
16 *+HasArrived(FireFighters). Happens(Start(FireContainement), 15).*
17 *+Started(FireContainement).*
...
25 *Happens(Arrives(CollectionCoordinator), 25).*
Happens(Arrives(CollectionManager), 25). Happens(Arrives(Librarian), 25).
26 *+HasArrived(CollectionCoordinator). +HasArrived(Librarian). ...*
...
35 *Happens(End(FireContainement), 35).*
36 *+Finished(FireContainement). Happens(Start(Collections*
CoordProcess), 35). Happens(Start(FacilitiesCoordProcess), 36).
Happens(Start(SalvageControllerProcess), 36).
37 *+Started(CollectionsCoordProcess). +Started(Facilities*
CoordProcess). +Started(SalvageControllerProcess).
Happens(Start(RecoveryProcess), 37).
38 *+Started(RecoveryProcess).*
...
46 *Happens(End(RecoveryProcess), 46).*
47 *+Finished(RecoveryProcess).*

The instantiated model shows that there exists a solution based on design level contracts,the left column shows the time points while the right column shows the event or process state (+ sign shows that a particular fluent holds at that time point). We will use this instantiated model as base for describing different aspects of the proposed monitoring framework.

V. Proposed Monitoring Framework

The proposed event-based monitoring framework allows to specify and reason about the monitoring properties during composition process execution. The composition process is specified using the event-calculus and is then used to instantiate, verify and execute the composition process (see Figure 3-①). The instantiation phase involves finding a solution to the composition process using the event calculus reasoner and the instantiated plan is then executed using the execution engine (see Figure 3-②).

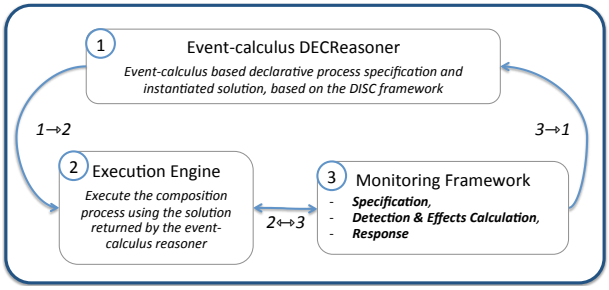

Figure 3. Proposed monitoring framework

The proposed monitoring framework (see Figure 3-③) works during the composition process execution and is divided into three phases. The *specification* phase requires the user to specify the functional and non-functional properties that needs to be monitored to identify anomalies or needed for KPI's measurement. Then, the detection and effects calculation phase is both responsible for detecting any violations based on the specified properties and to calculate the side-effects the detected violation has on the overall process. Then, the *response* phase uses the user-specified actions to respond to the monitored property. In the sections to follow, we will first discuss the monitoring properties specification in Section-VI and then will discuss how the detection and effects calculation works once a violation is detected, in Section-VII. Then, we will discuss the possible response actions to cater for the monitoring properties, in Section-VIII.

VI. Properties Specification

The *specification* phase requires the user to specify the functional and non-functional properties that needs to be monitored to identify anomalies or needs for KPI's measurement. The properties that need to be monitored are added to process description either at the process design (if they are already known – figure 3-①) or they can be added to the process specification at the execution time. In the later case the process specification is updated and an updated instantiated solution is sought, in order to verify any conflicts and to get an updated execution plan as a result of process change during execution (see Figure 3–*3→1*).

Properties that can be monitored include the functional aspects such as monitoring the invocation and execution order or they can be based on non functional aspects such as temporal aspects requiring to monitor the response time for a service, delay between successive invocations of the service or monitoring invocation time-frame for a service. Further, the properties can also be based on data such as monitoring the data availability, validity and expiry or based on the security properties such as monitoring the data integrity, confidentiality, access-control. The choice of highly-expressive event-calculus formalism even allows to combine the properties related to temporal, security and other aspects such as monitoring the data validity and access control within specific time frame which may be needed for instance, during dynamic task delegation (see [20] for details). Below we discuss event-calculus models for some of these properties:

Happens(StartInvoke(S1), time1) & Happens(EndInvoke(S1), time2) & time2 - time1 = SomeTimeValue → Happens(Terminateprocess(), time2).
Happens(InvalidateResponse(service), time) → Happens (SendAlertNotification(), time).

The first axiom in the model above specifies a monitoring property for monitoring the response time for a service. If the difference in the occurrence of the Start and End event is greater than SomeTimeValue, the process is terminated. The monitoring properties have the general form $Property \rightarrow Response$ and we will discuss different response actions later in Section-VIII. The second axiom above specifies to send Alert notification once the response message from any service does not remain valid. Space limitations restrict us to discuss event-calculus models for other aspects, a detailed discussion about how different security and temporal aspects can be modeled using event-calculus can be found in [3].

Regarding the motivating example, we first consider different monitoring properties already known at the composition design stage as identified in the motivating example. The event-calculus axioms for these properties have been added to the composition design, and thus the solution returned by the reasoner caters for these properties (see Figure 3-①). Then, as the proposed approach also allows to specify the monitoring properties during process execution, we consider the following properties added to the process specification once the response message from the fire-brigade and emergency staff calling services have been received.

- REQ-4: Arrival time of staff members (and completion of activities) should be logged.
- REQ-5: Salvage process should only be handled by

Collection manager as the Librarian does not possess the expertise to handle the degraded specimens. However in the presence and help from the conservator, he can handle the salvage process.

Adding the monitoring properties may require updating the process specification (see Figure 3–*3→1*), such as new axioms/event for logging should be added. The last axiom in the event-calculus model below handles the REQ-5 while other axioms handle REQ-4:

Activity LogArrival
Happens(Arrives(user), time) → Happens(Start(LogArrival), time).
Happens(Start(SalvageControllerProcess), time) → HoldsAt(HasArrived (CollectionManager),time).

VII. Detection and effects calculation

A. Detection

The detection of the violations can be handled at different levels using the proposed framework. At a basic level we first consider the violations to the execution plan, which is handled by maintaining an *event repository* which keeps track of all the messages exchanged between the composition process and the participating services during process execution. This repository is then used to find any mismatch between the temporal ordering of actual events and the ones mentioned in the initial instantiated plan. Using the basic detection technique, it is possible to find violations to the execution plan or the invocation and execution order of the services. However such a detection level may not be useful in detecting data values based or other low-level violations, as using the event-calculus, the process is modeled at an abstract level. This can be handled by also abstracting the processing of verifying the data values and other low level service details by using event-calculus fluents. For instance, we can have a fluent *ResponseValid(SomeService)* and an event called *ValidateResponse(SomeService)*, and whenever data is received from a service we check for its validity. Then, if the data is not considered valid, based on application level checks on data, the fluent *ResponseValid(SomeService)* does not hold and in-turn results in a mismatch between the initial instantiated plan and actual service execution. The detection phase may thus require the execution engine support (for instance checking data validity, see Figure 3-②).

Then, in order to detect the monitoring properties added at the execution time (e.g. based on external events not there in the initial instantiated plan), the ”abduction reasoning” mode can be used by adding the newly added events and monitoring properties to the process model and re-invoking the reasoner. In case of no conflict and violation, the reasoner returns an updated plan based on the added events and monitoring axioms. However, if there is some conflict based on addition of new events or if the newly added monitoring property is not satisfied, the reasoner returns a set of unsatisfied clauses highlighting the error. The detection phase may thus also require the reasoner support (see Figure 3–*3→1*).

B. Effects calculation

Once a violation to some monitoring property is detected, the effects calculation phase is responsible for calculating the side-effects this violation has on the overall process flow. This allows to prioritize the violations based on their impact and it may be possible to ignore some violations, for instance if the response time delay for a service has no effect on the overall process goal and other functional and non-functional properties associated with the composition process. As the proposed approach allows to reason about the composition process and as the approach is based on event-calculus with different reasoning modes, the effects calculation is achieved by adding the partial plan with the violation to the initial plan and re-invoking the reasoner. Although the process may seem similar to the detection of monitoring properties added at the execution-time, there is one major difference; instead of using the ”abduction reasoning” we use ”deduction reasoning” in the effects calculation phase. This may further allow to foresee any anomalies which may not be evident now but may happen later in the process execution. The effects calculation phase thus requires the support from the event-calculus reasoner to perform deductive reasoning (see Figure 3–*3→1*).

C. Example

Let us now review the monitoring properties identified earlier for the motivating example and discuss how any violation to these properties can be detected. The properties REQ-1 and REQ-3 are implicit and these properties are evaluated whenever the reasoner is reinvoked. For instance, the property REQ-1 specifies the composition goal and is evaluated every time the abductive reasoning is sought. Then, the property REQ-2 relies on application level data validity checks and requires to update the process specification based on data received from the participating Web services. The event added at the design-time, *ValidateAndUpdatePlan(service)*, models this behavior. Then, updating the process specification during execution may result in conflicts and inconsistencies, such as the axiom added for REQ-5 may contradict with the one already there in the process description. However, the proposed framework allows to identify and resolve these conflicts as adding these axioms and re-invoking the reasoner will either provide an updated solution (as for REQ-4) or will return the conflicting clauses. Adding property REQ-4 to the specification and invoke reasoner for ”abductive reasoning” returns the following updated model:

...
25 *Happens(Arrives(CollectionCoordinator), 25).*
Happens(Arrives(CollectionManager), 25). Happens(Arrives(Librarian), 25).
Happens(Start(LogArrival), 25)...

The model above now contains the events added for logging and instructs the execution engine to log the arrivals of support staff. Further, adding the monitoring property, REQ-5 and re-invoking the reasoner, returns the a set of

conflicting clauses including the following:

2708 -550 0: (HoldsAt(HasArrived(CollectionManager), 36) — !Happens(Start(SalvageControllerProcess), 36)).

The model above shows that adding the execution-time properties can result in a conflict, as at the design time it was specified that either the Librarian or Collection Manager can perform the role of SC and a solution exists at design time (showing Librarian can be there early and do the job). However, at execution time, the constraint has been modified to only allow Collection Manager to act as a SC and as he is not there, the process shows there is a conflict. However, the property REQ-5 allowed Librarian to do the task if Conservator is there. Let us consider that the Conservator does arrive in time, and re-invoking the reasoner gives following updated model:

```
...
35  Happens(Arrives(Conservator), 35).
    Happens(End(FireContainement), 35)
36  +Finished(FireContainement). +HasArrived(Conservator).
    Happens(Start(CollectionsCoordProcess), 36). Happens(Start
    (FacilitiesCoordProcess), 36). Happens(Start(SalvageControllerProcess), 36).
```

VIII. Response

The response for the monitoring properties may involve some domain specific actions to cater for or measure the KPI's and other parameters (such as logging, performance evaluation) needed for the successful process execution. Then, in order to cater for the monitoring violations detected at the execution time, different recovery actions can be used in-order to recover from the violation. These actions may include to ignore the violation, terminate the process, re-invoke or substitute the service, find an alternative solution based on current process state or backtrack to some previous state and then seek an alternative solution and others. Below we briefly discuss the alternative-path as a recovery action as it highlights the need for a reasoning-based approach.

The recovery process is handled by adding the current process state (with the violation) and re-invoking the reasoner to perform abductive reasoning for the goal. However, it is not always possible to recover from a violation *AND* respecting the associated constraints and composition goal. As a result, some constraints may require to be relaxed and the proposed approach allows to identify the conflicting clauses and hard-constraint if a recovery solution is not possible. The proposed approach thus preserves all the functional and non-functional constraints associated with the composition process (unless needed to be relaxed) while performing recovery. Further, the proposed approach allows both to find a new solution based on the current process state (thus specifying what steps should be taken now to recover from the violation and hence termed forward recovery) or to backtrack to some previous activity (if possible) and try to find a new from there. The response phase may require the execution run-time support (for instance actions such as logging, KPI's measurement, see Figure 3–*3*→*2*) and may also require the support from the DECReasoner in order to do abductive reasoning for actions such as finding alternatives (see Figure 3–*3*→*1*).

In relation to the motivating example, we will consider the case when the fire-containment process is taking more time than initially estimated. As a result, a violation is detected at time point and a recovery solution is sought requiring to find alternative path for successful process execution. As a result, the reasoner is re-invoked with the updated process state (with the violation and new estimate for the fire containment, it would probably take 30 minutes instead of 20 minutes, as planned) and we get the updated model as below:

```
...
35  Happens(Arrives(Conservator), 35).
36  +HasArrived(Conservator).
...
45  Happens(End(FireContainement), 35)
46  +Finished(FireContainement).
    Happens(Start(CollectionsCoordProcess), 46). Happens(Start
    (FacilitiesCoordProcess), 46). Happens(Start(SalvageControllerProcess), 46).
...
47  +Started(CollectionsCoordProcess). +Started(Facilities
    CoordProcess). +Started(SalvageControllerProcess).
    Happens(Start(RecoveryProcess), 47).
48  +Started(RecoveryProcess).
...
56  Happens(End(RecoveryProcess), 56).
57  +Finished(RecoveryProcess).
```

The updated model shows that even with the violation, the goal is still achievable satisfying all the associated constraints. However, had it not been the case, some constraints may need to be relaxed (if possible) to achieve the goal.

IX. Performance evaluation

In order to test our proposal, we have implemented the proposed model using the discrete event calculus language [21] and discussion about the implementation architecture for the **DISC** framework can be found in [2]. The proposed monitoring approach requires to reason about the composition process at various stages of the framework. In our previous work [2], [3] we concluded that although the solution finding by SAT solver is highly efficient, the event-calculus-to-SAT encoding process does not scale well with the increase in time points and complexity of the composition process. In this work, we have thus modified the encoding process by two approaches. First, the process encoding is done only once during the instantiation phase of the **DISC** framework and encoding for any subsequent changes to the process description, such as during process execution or during effects calculation phase of the proposed monitoring framework, is added to the initial process encoding. Then, we have modified the c language code for the encoding process (changes include modifying the hash function to have less collisions) and the results show a substantial gain in performance. The performance evaluation tests for the motivating example were conducted on a *MacBook Pro* Core 2 Duo 2.53 Ghz and 4GB RAM running *Mac OS-X 10.6*. The DECReasoner version 1.0 and the SAT reasoner, relsat-2.0 were used for reasoning. The performance evaluation

results are shown in Figure-4, with Y-axis showing the time-taken in seconds while the X-axis showing the problem size, which is obtained by multiplying the composition problem (with same dependencies) and increasing the time points. This results in a highly complex process needed to test the original and modified encoding process. The performance evaluation results show substantial gain in event-calculus to SAT encoding process. The solution computation by relsat solver is highly efficient and the recovery process always takes less (or same) time than the initial solution as we have a partial plan and that reduces the problem size.

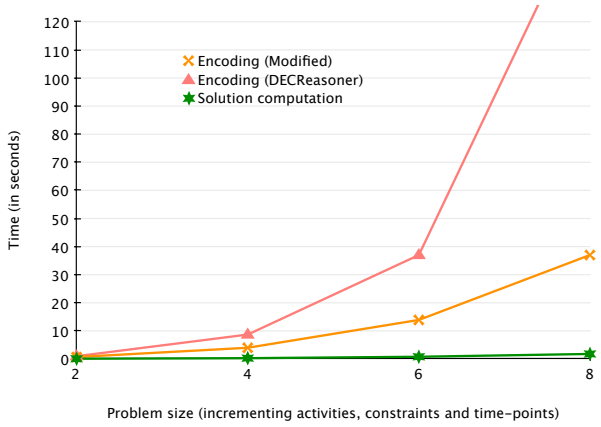

Figure 4. Performance evaluation results

X. Conclusion

In this paper, we have presented an integrated event-based framework that allows to specify and reason about the monitoring properties during process execution. The proposed approach builds upon an event-based declarative composition design [2] and does not require to define events as an extension to the composition design as the events are first class objects of both composition design and monitoring framework. The proposed approach is highly expressive and allows the specification of monitoring properties that are based on both functional and non-functional (such as temporal and security) requirements, that can be added to the process specification during design or process execution. Further, it allows for identifying any violations based on the specified properties and for calculating the effects any monitored violation has on the overall process execution. Then, any recovery solution takes care of the process specification and the associated QoS properties associated with the process. We have presented a real world crisis management case-study that highlights the proposed framework. Further, we have modified the encoding process for DECReasoner tool, as it does not scale well, and it results in substantial performance improvement.

References

[1] O. Moser, F. Rosenberg, and S. Dustdar, "Event driven monitoring for service composition infrastructures," in *WISE*, 2010, pp. 38–51.

[2] E. Zahoor, O. Perrin, and C. Godart, "Disc: A declarative framework for self-healing web services composition," in *ICWS*, 2010.

[3] ——, "Disc-set: Handling temporal and security aspects in the web services composition," in *ECOWS*, 2010.

[4] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Workflow exception patterns," in *CAiSE*, 2006, pp. 288–302.

[5] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[6] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through sese decomposition," in *ICSOC*, 2007, pp. 43–55.

[7] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser, "Automatic workflow graph refactoring and completion," in *ICSOC*, 2008, pp. 100–115.

[8] D. Ghosh, R. Sharman, H. R. Rao, and S. J. Upadhyaya, "Self-healing systems - survey and synthesis," *Decision Support Systems*, vol. 42, no. 4, pp. 2164–2185, 2007.

[9] R. Griffith, G. E. Kaiser, and J. A. López, "Multi-perspective evaluation of self-healing systems using simple probabilistic models," in *ICAC*, 2009, pp. 59–60.

[10] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni, "Exception handling for repair in service-based processes," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 198–215, 2010.

[11] C. Beeri, A. Eyal, T. Milo, and A. Pilberg, "Bp-mon: query-based monitoring of bpel business processes," *SIGMOD Record*, vol. 37, no. 1, pp. 21–24, 2008.

[12] L. Baresi, S. Guinea, O. Nano, and G. Spanoudakis, "Comprehensive monitoring of bpel processes," *IEEE Internet Computing*, vol. 14, no. 3, pp. 50–57, 2010.

[13] M. Sun, B. Li, and P. Zhang, "Monitoring bpel-based web service composition using aop," in *ACIS-ICIS*, 2009.

[14] G. Wu, J. Wei, and T. Huang, "Flexible pattern monitoring for ws-bpel through stateful aspect extension," in *ICWS*, 2008.

[15] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Run-time monitoring of instances and classes of web service compositions," in *ICWS*, 2006, pp. 63–71.

[16] L. Baresi, S. Guinea, M. Pistore, and M. Trainotti, "Dynamo + astro: An integrated approach for bpel monitoring," *ICWS*, pp. 230–237, 2009.

[17] K. Mahbub and G. Spanoudakis, "A framework for requirents monitoring of service based systems," in *ICSOC'04*, 2004.

[18] R. A. Kowalski and M. J. Sergot, "A logic-based calculus of events," *New Generation Comput.*, vol. 4, no. 1, 1986.

[19] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "Paws: A framework for executing adaptive web-service processes," *IEEE Software*, vol. 24, no. 6, 2007.

[20] K. Gaaloul, E. Zahoor, F. Charoy, and C. Godart, "Dynamic authorisation policies for event-based task delegation," in *CAiSE*, 2010.

[21] E. T. Mueller, *Commonsense Reasoning*. CA, USA: Morgan Kaufmann Publishers Inc., 2006.