An Integrated Declarative Approach to Web Services Composition and Monitoring

Ehtesham Zahoor, Olivier Perrin, and Claude Godart

LORIA, INRIA Nancy Grand Est Campus Scientifique,
BP 239, 54506, Vandoeuvre-lès-Nancy Cedex, France
{ehtesham.zahoor,olivier.perrin,claude.godart}@loria.fr

Abstract. In this paper we propose a constraint based declarative approach for Web services composition and monitoring problem. Our approach allows user to build the *abstract composition* by identifying the participating entities and by providing a set of constraints that mark the boundary of the solution. Different types of constraints have been proposed to handle the composition modeling and monitoring. *Abstract composition* is then used for instantiating the *concrete composition*, which both finds and executes an instantiation respecting constraints, and also handles the process run-time monitoring. When compared to the traditional approaches, our approach is declarative and allows for the same set of constraints to be used for composition modeling and monitoring and thus allows for refining the *abstract composition* as a result of run-time violations, such as service failure or response time delays.

1 Introduction

Traditional Web services composition approaches (such as WS-BPEL and WS-CDL) tackle the composition problem by focusing on the control flow of the composition process. Although control over the composition process is critical, in some cases it must be relaxed to some extent to make the process flexible, but flexibility and control on the composition process are conflicting requirements. We detail in this paper a sample crisis management scenario that highlights the importance of proper balance between control and flexibility of the composition process. A problem of traditional service composition approaches is that they are procedural and as proposed in [14] they over constrain the composition process making it rigid and not able to handle the dynamically changing situations. Further the focus on data, temporal aspects and other non-functional requirements is not thoroughly investigated. Another important aspect is the run-time monitoring of the composition process and although it is tightly coupled with the composition process, it is not well integrated to the traditional composition approaches. Proposed solutions introduce a new layer for the composition monitoring and thus does not provide the important execution time violations feedback to the composition process. Finally, the scalability of the composition process is an important factor as the number of available services to choose from

is increasing rapidly. As a result, exploring all possible solutions to the composition problem may not be a feasible option and some choices should be made at different stages to avoid the solution explosion of the composition process.

In this paper we propose a constraint based declarative approach for Web services composition and monitoring problem. Our approach allows user to build the *abstract composition* by identifying the participating entities and by providing a set of constraints that mark the boundary of the solution. Different types of constraints have been proposed to handle the composition modeling and monitoring. *Abstract composition* is then used for instantiating the *concrete composition*, which both finds and executes an instantiation satisfying constraints, and also handles the process run-time monitoring.

When compared to the traditional approaches, our approach is declarative and allows for the same set of constraints to be used for composition modeling and monitoring and thus allows for refining the composition model as a result of run-time violation. Moreover, our approach models both the data and control flow and the constraints include both the functional and non-functional specifications (such as security and temporal aspects on both control and data). Further, in contrast to the procedural approaches, we propose a declarative approach to model the composition process. Then, our approach aims to target the conflicting requirements of flexibility and control on the composition process. At one hand, user can loosely constrain the composition process to provide the composition engine the flexibility to choose the solution. On other hand user can over constrain the composition process to focus on the control. Further, to handle the scalability requirements our approach allows for one best matched (user chosen) Web service as a result of node instantiation and handles the case when the service selection choice needs to be backtracked based on dependency between services and allows for propagation of newly chosen solution.

2 Motivation and related work

The motivation for our work stems from the process modeling and monitoring in a crisis situation and we present a crisis management scenario that highlights the benefits of the approach. A crisis situation is, by nature, a dynamic situation especially in its first phases. It also demands for a composition that is characterized by temporal constraints, uncertainty, multiple and changing goals, coordination of multiple services and multiple data sources, and require the composition process to be more flexible to adapt to continuously changing environment. The situations these ad hoc compositions are dealing with are complex, ambiguous, and very dynamic. Information arrives from multiple sources, with varying degrees of reliability and in different formats. Information that was treated at time t may be superseded by new information at time t+1.

The interesting concept with a crisis scenario is that it brings together two related dimensions: *organization* and *situation*.*Organization* encompasses the design time composition modeling which involves identifying activities and control and data flow between them. There have been many approaches to model this

dimension. Most of these approaches can be divided into Workflow composition and AI planning based approaches, as discussed in [10]. The composition result can be regarded as a workflow because it includes the atomic Web services and the control and data flow between these services. *Static* workflow composition approaches require an *abstract composition* to be specified and the selection and binding is performed automatically by the Web services composition process, while the *dynamic* workflow composition approaches require to both build the *abstract composition* and select atomic service automatically based on user request as proposed in [11]. The composition process can also be regarded as a AI planning problem assuming that each Web service can be specified by its preconditions and effects (using situation calculus [5, 8], rule-based planning [6], theorem proving [15] or other approaches including [13]).

The problem of traditional approaches (such as WS-BPEL or WS-CDL) is that all what is not explicitly modeled is forbidden. These approaches have in common that they are highly procedural, i.e., after the execution of a given activity the next activities are scheduled. Seen from the viewpoint of an execution language their procedural nature is not a problem [14]. However, unlike the modules inside a classical system, Web services tend to be rather autonomous and an important challenge is that all parties involved need to agree on an overall global process. Moreover, this way of modeling renders difficult to model complex orchestrations, i.e. those in which we need to express not only functional but also non-functional requirements such as cardinality constraints (one or more execution), existence constraints, negative relationships between services, temporal constraints on data or security requirements on services (separation of duties for instance). With current approaches, the designer should explicitly enumerate all the possible interactions and in turn over-constrain the orchestration. In case of multiple constraints, the problem becomes even more difficult. Moreover, the flexibility of the obtained model is really low as modifying one aspect (e.g. temporal) has important side effects on other aspects (e.g. control flow or security). A more detailed discussion can be found in [9]. When compared to other declarative approaches [14, 9], our approach allows for the same set of constraints to be used for both process modeling and monitoring.

The second dimension a crisis situation focuses on is the *situation*. The composition process to handle the crisis should be able to measure and to adapt to continuously changing situation. This leads to the problem of Web services monitoring and the approaches for dealing with Web services monitoring include [1, 2, 4]. The problem with current monitoring approaches is that they are mostly proposed as a new layer to the procedural approaches such as WS-BPEL. As a result, they are unable to bridge the gap between organization and situation in a way that it is not possible to learn from run-time violations and to change the process instance (or most importantly process model) at execution time.

The need of observability (the feedback that provides insight of a composition), the support of dynamicity (ability to change resources, services, and ordering as situations change and evolve), the support of focus change (ability to reorient focus in a dynamic environment), and the support of various perspec-

tives (ability to consider the organization given different points of view - control, data,...) guide the motivation of our approach. We believe that the declarative approach appears to be well adapted rather than the traditional imperative approach. Using a declarative language allows to concentrate on the "what" rather than the "how" and it is more flexible as you specify only the boundaries of the composition rather than its precise execution (reducing the over-specification associated with the imperative method). Then, the monitoring of the composition is largely facilitated as the same constraints can be used for both the definition, the instantiation, and the execution of the composition.

3 Motivating example

Let us consider a sample scenario when the emergency landing of the plane carrying important government officials has resulted in serious injuries to the passengers. An emergency center has been set up in the remote region for handling patients. In a typical SOA based setup, the emergency center works by contacting the Web services provided by different systems. Depending on the condition of each patient the emergency center may either opt for nearby initial checkup center or for the detailed checkup center, for providing patient emergency treatment and to examine the nature of injuries to the patient. The emergency center may also decide to transfer the patient to some nearby hospital (not known in advance), this choice will be made using the Web services provided by different hospitals and will also be based on certain constraints such as the hospitalization and surgery facilities availability and some non-functional properties such as reliability, temporal requirements and others. The chosen hospital Web service can then be used to schedule operation theatre, allocate surgery team and to provide critical data to the hospital.

The composition process may also decide to discover and communicate with the ambulance service (or SAMU[1] service for serious injuries) to transfer the patient to the selected hospital and again, as the Web service is not known in advance, some constraints may be specified to discover the service.

Due to high-profile passengers, contacting Police department for assistance may be needed. Further the access to the patient information file from the Web service provided by the social security system may also be needed. Finally, the emergency center may also discover and contact some blood bank service to arrange additional blood supply for the patient (if patient blood type is rare).

4 Proposed framework

Our proposal aims to provide a declarative framework for addressing the Web services composition and monitoring problem, such as the one presented in the

[1] SAMU (Service d'Aide Médicale d'Urgence) is the French hospital based emergency medical service.

motivating example. In this section we will briefly discuss the main concepts related to our approach and will detail them in the sections to follow. Our proposed framework has two main stages, *abstract composition* and the *concrete composition*. Each stage has a set of constraints targeted to handle the organization (composition modeling) and the situation measurement dimension (monitoring). This gives our framework the flexibility to use the same set of constraints for bridging the gap between organization and situation measurement (see figure 1).

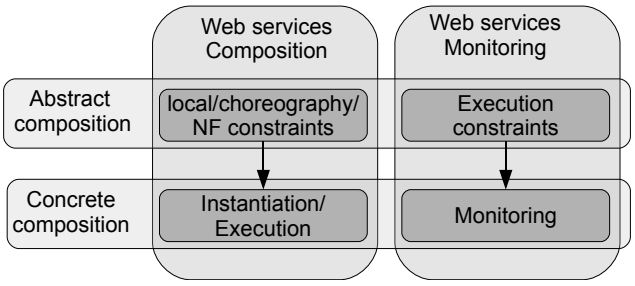

Fig. 1. Proposed framework components

The composition process starts when the user specifies the *abstract composition*, using a user friendly interface, allowing her/him to drag and drop components and provide constraints. Various related concepts include:

- *Web services* - The user can specify the concrete Web services instances known in advance, to be used within the composition process.
- *Nodes* - If the Web service instance is not known in advance, the user can specify the Web service node which has a unique type such as Hospital.
- *Constraints* - Constraints specify the boundaries for the solution to the composition process and different type of constraints can be added to the *abstract composition* process for handling modeling and monitoring dimensions. The constraints related to composition modeling include the *local,choreography* and *non-functional* constraints while the constraints for handling composition monitoring include *execution* constraints (see figure 1).

The *abstract composition* specified by the user is used to instantiate the *concrete composition* phase. As similar to the *abstract composition*, the *concrete composition* process also has different stages for handling composition modeling and monitoring. Local and choreography constraints are used for nodes instantiation and process execution while the execution constraints are handled at run-time monitoring phase of the *concrete composition*.

In order to model the *abstract composition*, our approach relies on the Event Calculus (EC) [3, 7]. 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 abstract composition 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).

The EC is a first-order logic that comprised the following elements: $\mathcal{A}$ is the set of *events* (or actions), $\mathcal{F}$ is the set of fluents (fluents are *reified*[2]), $\mathcal{T}$ 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 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.
- $Initially(f)$ - fluent f holds from time 0.
- $Clipped(t_1, f, t_2)$ - fluent f was terminated during time interval $[t1, t2]$.
- $Declipped(t1, f, t2)$ - fluent f was initiated during time interval $[t1, t2]$.

Further, some event calculus axioms are available that relate the various predicates together. Using EC, we are able to represent both the organization, i.e. the *abstract composition* of services, and the situation, i.e. the verification that everything goes as planned at execution time.

5 Abstract composition

5.1 Constraints

The constraints added to the *abstract composition* serve as the boundaries for the acceptable solution to the composition problem. These constraints can be divided into following categories:

- **Local constraints** are the constraints added to the Web service nodes in the composition process. These constraints specify the properties that should be respected while binding the Web service nodes to concrete Web service instances and as our approach aims to choose the best matched solution for the node instantiation, the local constraints specify one specific path (solution) to choose from all available paths (solutions) for the Web services composition process. Local constraints can be in the form of non-functional requirements such as security, reliability, quality requirements. They can also be in the form of some domain specific functional properties (hospitalization, surgery facilities availability for the motivating example). Formally, local

[2] Fluents are first-class objects which can be quantified over and can appear as the arguments to predicates.

constraints are translated as predicates in EC. For instance, service s_1 is reliable would be written with the following formula: $reliable(s_1, value)$ where *value* is true.

- **Choreography constraints** specify the constraints regarding the control flow of the composition process and express the order and execution sequence of the participating activities. Some examples of choreography constraints include *before*, *after*, *if-then-else*, *choice* and others. Choreography constraints are also guided by the dependency between the participating entities, specifying that a service s_1 has a dependency on service s_2 will require s_1 to be executed before the service s_2. Formally, following EC formula specifies that service s_1 must be executed before service s_2:
  $Initially(forbidden(s_2, f)) \wedge Terminates(s_1, forbidden(s_2, f), \_)$.
- **Non-functional constraints** specify the constraints independent of the functional aspect of the web service composition. It can be for instance security requirements. Formally, if we want to model a specific security rule stating for instance that once a service s_1 has been executed, the service s_2 cannot be executed for the next 20 minutes, we write:
  $Initiates(s_1, forbidden(s_2, f), t_1) \wedge Declipped(t_1, forbidden(s_2, f), t_2) \wedge (t_1 + 20 \leq t_2)$.
- **Execution constraints** specify the constraints to be validated at run-time. These constraints take the form of *monitors*, which have a associated monitoring event/condition and actions to perform if the condition to be monitored is encountered. We will take a detailed look on monitors in section-7

5.2 Example

Let us now review the motivating example and discuss how the *abstract composition* can be specified, introducing the associated constraints. The *abstract composition* is specified using abstract-refine approach, the base *abstract composition* for the motivating example can be specified as:

initialCheckupWS, detailedCheckupWS, Hospital(?h), Ambulance(?a), SAMU(?s), regionalPoliceWS, someSocialSecurityWS, BloodBank(?b)

The presentation syntax above is used to describe the participating Web services in the composition process, the question mark (?) operator marks the variables, i.e. the Web service nodes that have not yet instantiated. The syntax also specifies the type of participating nodes.

The base *abstract composition* has no constraints added to it, i.e. all that is specified is the invocation (or instantiation and invocation for Web service nodes). To mark the boundaries of the abstract composition, we start by adding the different type of constraints to the *abstract composition*. The *initialCheckup* is a concrete (already known) web service and thus has no local constraints. For the choreography constraints, we consider the service to be invoked *before* the *hospital* and *detailedCheckup* service and that if the *initialCheckup* service is executed then the *detailedCheckup* service is also executed. Further, the service has data dependency on the patient information from the *socialSecurity* Web service. As part of execution constraints, we consider that the data validity from

the service is for 1 hour and the response time of the service should be less that 5 ms. Finally, for the cardinality constraints, which are part of the choreography constraints, we consider that the service can be executed zero or one times during the process execution. This marks the service to be optional and thus can be skipped for some instance (for all other participating services cardinality is exactly one). Below we present the event calculus formalization for the associated constraints to the *initialCheckupService* (ICS), we will detail the execution constraints later in section-7:

Choreography constraints:

before hospital - $Initially(forbidden(Hospital(h), true)) \wedge Terminates(ICSInvoked, forbidden(Hospital(h), true), \_)$

If ICS then detailedCheckup - $Initiates(ICSInvoked, HoldsAt(invoke(detailedChekup, true), t_2), t_1) \wedge t_1 < t_2$

data dependency on socialSecurityWS - $Initially(forbidden(ICS, true)) \wedge Terminates(socialSecurityWSInvoked, forbidden(ICS, true), \_)$

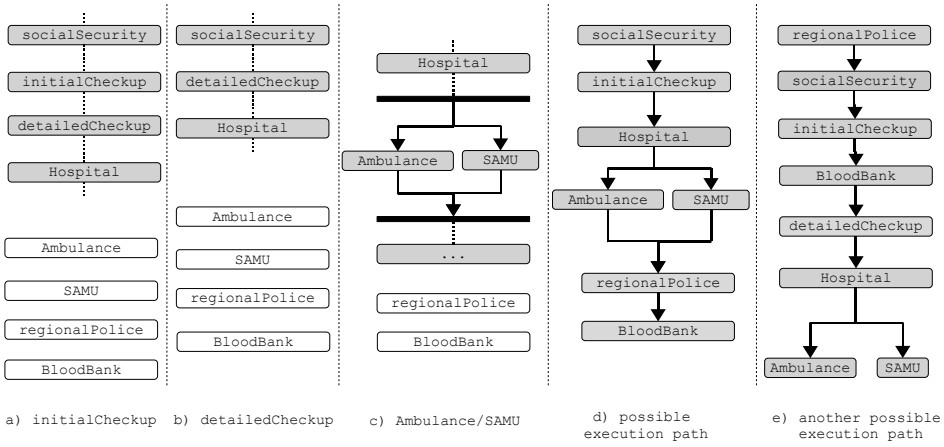

Fig. 2. Abstract composition for the motivating example

The choreography constraints associated with the *initialCheckupWS* also guide the partial control structure of the composition process (see figure 2-a), specifying constraints such as *before hospital* does not mark that the *hospital* service will immediately follow but specifies that there may be zero or many steps (services) between them. We can then have the similar constraints for the *detailedCheckup* service, excluding the constraint the if *detailedCheckup* is executed then the *initialCheckup* in also executed. This refines the partial control flow induced by the *initialCheckup* service (see figure 2-b). Next, for the hospital node we have some local constraints such as reliable, secure Web service and that the selected hospital must provide hospitalization and surgery facilities. Further

the hospital service has data dependency on initialCheckup and detailedCheckup Web services, below we present constraints modeling using EC, we will leave the discussion of execution constraint (in case of service failure re-instantiate hospital node) until section-7 :
Local constraints: $reliable(Hospital(h), true) \wedge providesSurgery(Hospital(h), true)$
Choreography constraints: Similar to EC model for *initialCheckup* service

For the ambulance and SAMU nodes, we have some local constraints and the choreography constraint that they cannot coexist. Further, they have data dependency constraint on the Hospital node and this refines the partial control flow by stating that in any solution to the composition process either of two services should be chosen after the hospital service invocation (see figure 2-c).
Local constraints:
$reliable(Ambulance(a), true) \wedge providesAirServices(Ambulance(a), true)$
Choreography constraints:
$Initiates(ambulanceServiceInvoked, forbidden(SAMUService, true), \_) \wedge$
$Initiates(SAMUServiceInvoked, forbidden(ambulanceService, true), \_)$

Then, for the *BloodBank* node, we can have some local constraints for service discovery and it has data dependency on the *socialSecurity* service. Finally, *regionalPolice* service is unconstrained and this gives the flexibility to invoke the service anywhere in the composition process. These constraints mark the boundary of the possible solution to the composition process but intentionally do not over-constraint the composition process providing the flexibility for process execution (see figure 2-d and 2-e for possible execution paths).

6 Concrete composition

The event calculus model for the abstract composition specified by the user can then be used to instantiate the concrete composition using the event calculus reasoner, below we highlight the various related concepts. The *concrete composition* process is divided into three phases; the *instantiation* phase handles the instantiation of Web service nodes to concrete Web service instances. The *execution* phase follows, which executes the instantiated Web services composition process, finally the *monitoring* phase handles the composition process monitoring during execution. In this section we will detail the instantiation phase and in the next section will discuss the monitoring phase of the composition process.

Instantiation The instantiation process is responsible for binding the Web service nodes to concrete Web service instances. The process starts by using the *local constraints* added to the *abstract composition*, that highlight the user preferences for the Web service discovery. These constraints are used to query the Web services repository for identifying the services satisfying these constraints however, in case of a loosely constrained node, the result set can be very large. Our proposal thus aims to choose the best matched Web service either selected

manually by the user or based on some user-specified criteria such as the quality rating for the Web service, by assuming that some trusted third-party has quality ratings assigned to services. For the instantiation process, we may also have to consider the choreography constraints associated to a Web service node in order to identify if the service has data dependency on the some already instantiated node. This will further constrain the Web service node to consider only the instantiations that respect the dependency between nodes. For the motivating example, the hospital node has data dependency on the initialCheckup service and thus may require to consider only the hospital Web services which can handle compatible data, this leads to a set of service composability rules which space limitations restrict us to detail.

If the instantiation result set for a node is empty then we have following possibilities. If some constraint is unsatisfied, user can be given option if she/he wants to relax the constraint. For example the user can decide to relax the reliability constraint in an attempt to discover new instances. Further, if the dependency between nodes is unsatisfied, we need to backtrack to the results of previous node to select some other instantiation solution and then proceed to finding solution for the current node. The process continues until all backtrack solutions have been explored. Finally when none of above two situations hold, the composition process fails with notifying the user of the intermediate results.

Then, an important aspect of our proposal concerns the ability for the instantiation to be modified at execution. Let us consider for instance that a node has been statically instantiated. At runtime, if the service fails, the node can be re-instantiated with a new service in order to continue the execution.

Backtracking The backtracking process involves finding an alternative to some previously chosen node instantiation solution. Backtracking is needed when the dependency between nodes is unsatisfied resulting in empty result set.

For the motivating example, the Ambulance node has data dependency on the Hospital node and lets consider the Hospital node has been instantiated to *someHospitalService* providing data in JSON format, then instantiating the Ambulance node will require us to consider only the Ambulance services requiring data in JSON format. Further, consider that there is no service available for the Ambulance node which can handle JSON data (however all can handle XML), this will require us to backtrack to the Hospital node to choose some other service, say *someOtherHospitalService* which may be providing XML data.

Propagation Once the backtracking process execution terminates, resulting in a newly chosen solution (instance), the composition solution must be recomputed and may require the propagation of newly chosen solution. This would likely be the case when a (partial) solution to the composition process has already been determined and backtracking to some higher node (in hierarchal order) may result in propagating the new solution. Further, propagation may also be needed when the user fine tunes the solution by manually selecting some other Web service after the instantiation process. In reference to the motivating example

scenario discussed for the backtracking process, the reinstantiation result i.e. *someOtherHospitalService* should be propagated to the Ambulance node.

7 Composition monitoring

The composition monitoring phase works by using the execution constraints, called monitors, attached to the abstract composition. Below we first briefly discuss the Event Processing Network (EPN) framework on which we will base our proposed monitoring framework.

7.1 Event Processing Network

Event processing network[12] is defined to be a pattern promoting the production, detection, consumption and reaction to events. An EPN model consists of four components, event producers (EP), event processing agents (EPA), consumers (C) and connection channels, called event channels (EC), for communication between other components. The EPA has following three stages, Pattern detection - responsible for selecting events matching a particular pattern, Processing - for applying processing functions to events detected and thus resulting in derived events and Emission - for emission of derived events.

Regarding proposed framework, the events generated by the composition process include the process startup, termination, and messages exchanged between the composition process and the services. Each event has associated header information which indicates the event meta-data including its source, type (such as inputMessage, outputMessage), time stamp and other similar information. In context of our proposed model, the composition process and participating Web services can be termed as the *event producers*. The produced events will then be processed by the EPA, which in our case is the *event listener* attached to the composition process.

7.2 Monitors

Monitors specify the execution constraints added to the abstract composition and each monitor has a set of activation conditions and associated actions.

Activation condition Each monitor has a set of activation conditions and the associated actions. The monitor activation conditions are based on the pattern detection stage of the EPN, below we discuss different activation stages.

- *Context* specifies the context of events that will be used for evaluating the event conditions. *Temporal* context can be specified to handle the conditions where monitoring is based on invocation history, as an example consider that we need to monitor the average response time for a Web service in last 24 hours. *Spatial* context can be specified, for example to monitor events originating from Web service in certain geographical region. Finally *semantic*

based context can be used to handle cases when generated events have relevance through mutual object or entity, as an example consider the case when we are willing to monitor the response time of all the Web services related to (or have the same type of) a particular Web service. The context can also be specified as of value *null*, requiring all the events to be processed.

- *Policies* can include decisions to either use first, last or each of event (within specified context) in stream for pattern detection. They can also apply further constraints to only include the events satisfying a predicate on their attributes or by specifying expiry time for events.

  As an example consider that only output messages from some service s, should be used for monitoring; we can thus specify the policy to consider all the events having type as *message* and *source* as service s in their meta-data.
- *eventConditions* as similar to the patterns in an EPN model, the events conditions specify the conditions to be checked for events conforming policies and that are within specified context. Event conditions are specified using event calculus and some common types of event conditions include verifying data values within messages being sent and received by the composition process, overall time taken by the composition process and others.
- *Directives* specify the directives for reporting monitoring violations to the actions stage. The monitoring process may decide to report the monitoring violations as they are observed by specifying the directive as *immediately*, this would likely be the case of a service failure. However, the monitoring process can also decide to delay the reporting by specifying directive as *delay - timeValue* to delay the reporting in an attempt to give the service some time to recover from the violation, this would likely be the case of exceeding response time for the Service.

Actions Once the eventCondition specified for the monitor is satisfied, associated actions specify the actions to be taken. Some common actions include terminate/ignore/reinstantiate and others. The re-instantiation function has an important application to the Web services monitoring process. In case of a service failure or a tardy service having a significant delay in the response time, to a service chosen as the result of the instantiation process, the monitoring process can add directives to re-instantiate the Web service node. The current service is added to the set of already used services for the node and a newly chosen service can then be used. This leads to the run-time composition of the Web services and a detailed discussion is beyond the scope of this paper.

The re-instantiation and then propagation to dependent nodes can be expensive, if the services are already in execution but it prevents the complete failure. Another important aspect is the handling of (partial) execution results of the service; if the service hasn't yet been invoked and no data is available, the reinstantiation is safe. If there are some intermediate result, they can either be discarded or passed to the newly instantiated service.

7.3 Example

Let us now review the motivating example and see how composition monitoring works using proposed framework. For the *initialCheckup* service we can specify some execution constraints that the data validity period for the service response is one hour and that the response time for the service should be less than 2 seconds. For the response time, the monitor below can be attached to the composition process as part of execution constraints of the service.

```
monitor: initialCheckup_responseTime
activation:
  context: none - every event should be taken into account
  policies: last request/response message, service = initialCheckup
  eventCondition:
  HoldsAt(requestSent(ICService, true), t) ∧
  HoldsAt(responseReceived(ICService, true), t′) ∧ t + 2 > t′
  directives: immediate
action: send response time alert message
```

The monitor above, added to abstract composition, can then be used by the event listener attached to the composition process for runtime handling. The monitor requires event listener to listen for the messages sent/received by the composition process that are within specified context (specified as none and thus listens for every event) and those conforming policies (last request/response messages from the *initialCheckup* service). Once the messages within specified context and conforming to policies is detected by the event listener, event conditions are checked (time difference between request/response messages greater than 2 seconds) and in case of conditions become true, directives (immediately) are observed to identify when to report this violation to actions stage. The actions stage can then take the appropriate actions (send notification message). Then for monitoring the data validation, the monitor below can be specified:

```
monitor: initialCheckup_dataValidation type: Message
activation:
  context: none
  policies: last response message, service = initialCheckup
  eventCondition:
  Declipped(t, available(checkupWSData, true), t′) ∧ (t + 60 > t′)
  directives: immediate
action: send data expiry alert message
```

Every service that is using the data of *initialCheckup* can then have a monitor to listen for the data expiry message of *initialCheckup* service and to take the corresponding actions. We can then also have a monitor to handle the Hospital node re-instantiation in case of service failure, omitted due to space limitations.

8 Conclusion

In this paper, we present a constraint based declarative approach for Web services composition and monitoring problem. Our approach allows user to build the *abstract composition* by identifying the participating entities and by providing a set of constraints that mark the boundary of the solution. Different types of

constraints have been proposed to handle the composition modeling and monitoring; local, choreography and non-functional constraints guide the composition design, while the execution constraints, called monitors and are based on Event Processing Architecture, are used for process monitoring during execution.

The *abstract composition* can then be used for the *concrete composition*, which involves instantiating the Web service nodes to the concrete Web service instances respecting local constraints associated with the nodes. The instantiation process then executes the composition and attaches the event listener to the composition process for handling run-time monitoring based on execution constraints. We have also presented a sample Crisis Management scenario, that highlights our approach.

References

1. F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS*, pages 63–71, 2006.
2. L. Baresi, S. Guinea, R. Kazhamiakin, and M. Pistore. An integrated approach for the run-time monitoring of bpel orchestrations. In *ServiceWave*, pages 1–12, 2008.
3. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.
4. K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS*, pages 257–265, 2005.
5. S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *KR*, pages 482–496, 2002.
6. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB J.*, 12(4), 2003.
7. M. Montali, F. Chesani, P. Mello, and P. Torroni. Verification of choreographies during execution using the reactive event calculus. In *WS-FM2008*, 2008.
8. S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.
9. M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops, Austria*, 2006.
10. J. Rao and X. Su. A survey of automated web service composition methods. In *SWSWPC*, pages 43–54, 2004.
11. H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and composing service-based and reference process-based multi-enterprise processes. In *CAiSE*, 2000.
12. G. Sharon and O. Etzion. Event-processing network model and implementation. In *IBM Systems Journal*, 2008.
13. E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, pages 17–24, 2002.
14. 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.
15. R. J. Waldinger. Web agents cooperating deductively. In *FAABS*, 2000.