In this paper, we propose an event-oriented framework called DISC that is both declarative and serves as a unified monitoring are thus active and widely studied research directions. However, the traditional approaches are both procedural (and rigid) and do not address the need of integrating these related dimensions using a unified formalism. In this paper, we propose an event-oriented framework called DISC that is both declarative and serves as a unified framework to bridge the gap between the process design, verification and monitoring. It provides a flexible and highly expressive composition design that can accommodate various aspects such as data relationships and constraints, Web services dynamic binding compliance regulations, security or temporal requirements etc. Furthermore, the DISC framework allows for instantiating and verifying the composition design and for monitoring the process while in execution.

Abstract

Web services are defined to be the software systems that provide interoperable machine-to-machine interaction over a network. Individual services may need to be composed and the composition process design, verification and monitoring are thus active and widely studied research directions. However, the traditional approaches are both procedural (and rigid) and do not address the need of integrating these related dimensions using a unified formalism. In this paper, we propose an event-oriented framework called DISC that is both declarative and serves as a unified framework to bridge the gap between the process design, verification and monitoring. It provides a flexible and highly expressive composition design that can accommodate various aspects such as data relationships and constraints, Web services dynamic binding compliance regulations, security or temporal requirements etc. Furthermore, the DISC framework allows for instantiating and verifying the composition design and for monitoring the process while in execution.

Keywords: SOA, Web services composition, Declarative, Integrated, Verification, Event-calculus, Self-healing

1. INTRODUCTION

Web services are in the mainstream of information technology and are paving way for inter and intra organizational application integration. High-level languages such as WS-BPEL and specifications such as WS-CDL and WS-Coordination extend the service concept by providing a method of defining and supporting orchestration (composition) of fine-grained services into more coarse-grained value added processes. The Web services composition process has different life-cycle stages. First, the process designer needs to model the composition process by using the fine-grained services to define new added-value processes. Then, the composition process needs to be verified to identify any anomalies and conflicts (such as deadlocks) in the process specification before execution. However, 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 etc. This highlights the need to monitor and detect the errors and accordingly react to cater for them.

Web services composition is a highly active research direction and in the literature, a number of approaches have been proposed that handle different stages of process life cycle. However, the traditional approaches have two major short-comings; first these approaches focus only on some stages of process lifecycle and this lack of integration results in a complex model such as mapping the BPEL based process specification to a particular automata with guards and using SPIN model checker for verification; BPEL to timed automata and using UPPAAAL model checker (Guerrouche & Godart, 2009b) for checking temporal properties. The lack of integration leads to the approaches that do not allow to learn from the runtime failures to provide the recovery actions, such as re-planning or alternative path finding, to recover from the monitored runtime violations based on current state of the process. Further, the proposed approaches aim to build on top of the traditional approaches (such as BPEL, OWL-S) which focus on the control flow of the composition using a procedural approach and as pointed out in (van der Aalst & Pesic, 2006), they over constrain the composition process making it rigid and not able to handle the dynamically changing situations. Furthermore, the focus on data, temporal, security aspects and other nonfunctional requirements is not thoroughly investigated. Adding these aspects makes the process very complex, as again they are proposed as a new layer on the top of existing BPEL based processes. The proliferation of partial solutions, the lack of expressiveness and simplicity to handle both functional and non-functional aspects, the lack of integration, the lack of recovery actions and the lack of flexibility mark the motivation for our work.

In this paper, we propose a declarative event-oriented framework, called DISC (Declarative Integrated Self-healing web services Composition), 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, Figure 1. Specifically, our contributions include:

**Figure 1. Proposed framework stages**

- **Expressive composition design:** The proposed framework allows for an EC-based composition design that can accommodate various aspects such as partial or complete process choreography and exceptions, different control flow constructs, data relationships and constraints, Web services dynamic binding, compliance regulations and other non-functional (such as different temporal and security) aspects.

- **Extensible approach:** The EC allows for integrating the existing work on composition design (Cicekli & Yildirim, 2000), authorization (Bandara, Lupu, & Russo, 2003; Gaaloul, Zahoor, Charoy, & Godart, 2010), and work on modeling other related aspects. Different EC models, presented in this work, are organized into independent generic patterns that can be added to EC-based process specification facilitating reuse and extension.

- **SAT-based process verification:** We advocate the need for the satisfiability solving for the verification of declarative processes. We have also proposed filtering criteria, based on the proposed patterns and on the structure of conflict clauses that can help to identify the clauses of interest. The proposed filtering approach is generic and can be applied to any problem specified using EC.

- **Event-based monitoring:** The proposed monitoring approach is built upon an event-based declarative composition design and this result in an integrated approach that allows to reason about the events. It does not require defining and extracting events from process specification, as the events are first class objects of both design and monitoring frameworks.

- **Implementation support:** The generic pattern-based approach allows us to implement a Java-based application, called ECWS, which allows abstracting the EC models from the process-designer and automates the composition process specification and verification. We have modified the DECReasoner code to gain substantial performance improvement as evident in performance evaluation results.

### 2. Motivation and Related Work

The motivation for our work originates from the need for process modeling, analysis and monitoring in a crisis situation. A crisis situation is highly dynamic. It demands for a process that is possibly partially defined. It is characterized by temporal and security constraints and uncertainty, and multiple and possibly changing goals. Hence, it 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. Most of the proposed approaches for the composition design can be divided into Workflow composition and AI planning based approaches. A problem of traditional Workflow oriented approaches (such as WS-BPEL and WS-CDL) is that they over-constrain the process that must be specified with the exact and complete sequence of activities to be executed. Although, this adds a lot to the control over the composition process, this control comes at the expense of process flexibility, making the process rigid to adapt to continuously changing situations, a detailed discussion can be found in (Pesic & van der Aalst, 2006). Furthermore, the traditional approaches make it 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 and security requirements on services. Some declarative approaches such as (van der Aalst & Pesic, 2006; Quartel et al., 2007) allow for defining process in a flexible way, but our approach allows for the same set of constraints to be used not only for composition modeling and analysis, but also for monitoring and violations feedback to composition design.

The design time verification of the composition process before execution is also an important aspect and a large number of proposed approaches require the mapping of composition process to some formal logic to be then used for verification. These approaches include mapping the BPEL process to particular automata with guards, and using SPIN model checker (Fu et al., 2004), BPEL to timed automata and using UPPAAL model checker (Guermouche & Godart, 2009b), BPMN constructs into labeled petri-nets then using BPEL2PLML (Dijkman, Dumas, & Ouyang, 2007) and others. Some approaches do consider the prospect of building upon a composition design that can be verified, such as Petri-net based design and verification as proposed in (Yi & Kochut, 2004), the process algebra based approach to compose and verify the composition process and the restricted abstract BPEL process to analyze the correctness.
are first class objects in both composition design and actions such as re-planning or alternate path finding as we approach allowing for effects calculation and recovery execution time, and it does not allow for a reasoning difficulty to learn from run-time violations and to change the approach for process specification does not bridge the gap between organization and situation in a way that it is very limited to design-time verification.

The second dimension a crisis situation focuses on is the situation measurement. The crisis handling composition process should be able to measure and adapt to continuously changing situation. This leads to the problem of Web services monitoring (Barbon, Traverso, Pistore, & Trainotti, 2006; Baresi, Guinea, Pistore, & Trainotti, 2009; Mahbub). The problem to effectively monitor and recover from the anomalies during process execution spans different related domains. Workflow management systems, in general, rely on an exception handling approach (Russell, van der Aalst, & ter Hofstede, 2006; Vanhatalo, Volzer, Leymann, & Moser, 2008). For the self-healing systems a number of approaches have been proposed (Ghosh, Sharman, Rao, & Upadhyaya, 2007; Griffith, Kaiser, & Lopez, 2009). In this paper, we would consider monitoring and recovery mechanisms for only services based processes, a detailed discussion about monitoring and recovery in different systems can be found in (Friedrich, Fugini, Mussi, Pernici, & Tagini, 2010).

Traditional composition process monitoring approaches, such as BPEL, build upon the exception handling approach for errors handling and allows to define different types 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 runtime and are tightly coupled and limited to it (Beeri, Eyal, Milo, & Pilberg, 2008; Baresi, Guinea, Nano, & Spanoudakis, 2010; Sun, Li, & Zhang, 2009; Wu, Wei, & Huang, 2008). As a result, they do not consider other sub-systems and processes that can be used for monitoring (Moser, Rosenberg, & Dlustdar, 2010).

In addition, the traditional monitoring approaches (Barbon et al., 2006; Baresi et al., 2009; Moser et al., 2010; Ardagna, Comuzzi, Mussi, Pernici, & Plebani, 2007; Friedrich et al., 2010) build upon composition frameworks that are highly procedural, such as BPEL, and this in-turn poses two major limitations. Firstly, 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 (Zahoor, Perrin, & Godart, 2010a). 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. A more detailed discussion can be found in (Zahoor E., 2011).

The growing need for incorporating the security and temporal aspects in the services composition has led to many approaches that aim to handle security and temporal aspects in the services composition at different levels of process lifecycle. In general, the proposed approaches are either based on the procedural approaches such as BPEL, lack expressiveness and ease to model complex temporal and security aspects (or most importantly, their combination) or mostly focus on only part of the problem. The proposed approaches for incorporating temporal aspects include (Guermouche & Godart, 2009a; Ponge, Benattallah, Casati, & Toumani, 2007) however, these approaches do not address the need for a unified framework and only focus on part of the problem. The proposed approaches also include (Kazhamiakin, Pandya, & Pistore, 2006), in which authors introduced a formalism called WSTTS to capture timed behavior of Web services and then using this formalism for model-checking WS-BPEL processes. The approaches also include ISDL (Quartel, Dijkman, & van Sinderen, 2004), which uses time attributes to represent properties. In order to verify the timed properties authors proposed converting the BPEL process specification to timed automata and using UPPAAL model checker (Guermouche & Godart, 2009b).

There have been many approaches that aim to handle the security aspects in the Web services composition (Menzel, Thomas, & Meinel, 2009; Garcia & de Toledo, 2008; Souza et al., 2009) however, as similar to the approaches for incorporating the temporal aspects, they only focus on a part of the problem.

The approaches that deal with the representation of the security aspects and aim to incorporate the security requirements into the business process definition include (Menzel et al., 2009; Neubauer & Heurix, 2008; Rodriguez, Fernandez-Medina, & Piattini, 2007). Furthermore, there have been approaches that aim to incorporate security requirements in the executable composition (Garcia & de Toledo, 2008; Chollet & Lalanda, 2008) or their enforcement at execution time (Song, Sun, Yin, & Zheng, 2006; Menzel et al., 2009). In (Souza et al., 2009), the authors proposed to use a formalism that allows for incorporating security aspects at different levels of abstraction, however the approach is procedural as it is based on and extends BPMN notations, lacks formal representation and does not allow for verification of and reasoning about the security properties.

The proliferation of partial solutions, the lack of expressiveness and simplicity to handle both functional and non-functional aspects, the lack of integration, the lack of recovery actions and the lack of flexibility mark the motivation for our work.

http://hipore.com/ijsc
3. COMPOSITION DESIGN

The composition process modeling is the first and most important stage of the composition process life cycle, Figure 1. The objective of composition process modeling is to provide high-level specification independent from its implementation that should be easily understandable by the process modeler who creates the process, the developers responsible for implementing the process, and the business managers who monitor and manage the process. The proposed DISC framework allows for a composition design that can accommodate various aspects such as partial or complete process choreography and exceptions, data relationships and constraints, Web services dynamic binding, compliance regulations, security and temporal requirements or other non-functional aspects. The composition design (and so are the other phases of the proposed framework) is based on EC.

3.1 EVENT CALCULUS (EC).

In order to model the composition design, our approach relies on the EC (Kowalski & Sergot, 1986). EC is a logic programming formalism for representing events and their effects and can infer “what is true when” given “what happens when” and “what actions do”, Figure 2. The “what is true when” part both represents the state of the world called initial situation and the objective or goal. The “what actions do” part states the effects of the actions. The “what happens when” part is a narrative of events. A detailed presentation can be found in (Zahoor et al., 2009).

The EC comprises the following elements: A is the set of events (or actions), F is the set of fluents (fluents are reified), T is the set of time points, and 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 EC predicates that are used for modeling the proposed framework are:

\[
\begin{align*}
\text{Initiates}(e, f, t) &\quad \text{fluent } f \text{ holds after timepoint } t \text{ if event } e \text{ happens at } t. \\
\text{Terminates}(e, f, t) &\quad \text{fluent } f \text{ does not hold after timepoint } t \text{ if event } e \text{ happens at } t. \\
\text{Happens}(e, t) &\quad \text{true iff event } e \text{ happens at timepoint } t. \\
\text{HoldsAt}(f, t) &\quad \text{true iff fluent } f \text{ holds at timepoint } t.
\end{align*}
\]

The choice of EC is motivated by several reasons. First, EC integrates an explicit time structure independent of any sequence of events (possibly concurrent). Then, given the composition design specified in the EC, a reasoner can be used to instantiate the composition design. 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 inductive tasks. In reference to our proposal, at composition design stage “abduction reasoning” can be used to find a set of plans or to identify any conflicts and at the composition monitoring stage, “deduction reasoning” can be used to calculate the effect of run-time violations. Figure 3 shows the mapping of EC to the proposed framework.

<table>
<thead>
<tr>
<th>What happens when</th>
<th>Specified initial orderings (partial plan, if any) and the sought execution plan for the specified goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>What actions do</td>
<td>Actions such as service invocation, nodes binding and associated effects</td>
</tr>
<tr>
<td>What is true when</td>
<td>The composition design including the Initial situation such as constraints, dependencies and the goal for the composition process (If any)</td>
</tr>
</tbody>
</table>

Figure 3. EC for the proposed framework

The EC models are presented using the discrete EC language (Mueller, 2006). We will only present the simplified models that represent the core aspects, intentionally leaving out the supporting axioms. All the variables (such as service, time, node etc) are universally quantified and in the case of existential quantification, it is represented with variable name within curly brackets, \{variablename}. For spacing issues, we abbreviate Response to Resp and Service to Serv.

3.2 COMPONENTS.

The various components that constitute the composition design can be broadly divided into activity and service categories. Activity is a general term for any work being performed. The services include either the Web services instances already known or abstract Web services (called nodes) that need to be instantiated (discovered) based on some specified constraints.

Activities. Each activity can have an activity life cycle as it changes states from being started till its completion. In order to model the activities using EC we can define events that
represent the actions required to start and finish the activities and defining EC fluents can represent the activity state. The EC based model for representing activities is shown in Figure 4.

**Figure 4. Activities model (with states)**

| sort activity | fluent Started(activity), Finished(activity) |
| event Start(activity), End(activity) |
| Initiates(Start(activity), Started(activity), time) |
| Initiates(End(activity), Finished(activity), time) |
| Terminates(End(activity), Started(activity), time) |

\[ \text{Happens(End(activity), time) } \Rightarrow \text{HoldsAt(Started(activity), time)} \]
\[ \text{HoldsAt(Started(activity), time) } \Rightarrow \neg \text{Happens(Start(activity), time)} \]
\[ \text{HoldsAt(Finished(activity), time) } \Rightarrow \neg \text{Happens(End(activity), time)} \]
\[ \text{HoldsAt(Finished(activity), time) } \Rightarrow \neg \text{Happens(Start(activity), time)} \]
\[ \neg \text{HoldsAt(Started(activity), 0)} \]
\[ \neg \text{HoldsAt(Finished(activity), 0)} \]

In the model as shown in Figure 4, we first define an EC sort named activity. The instances of the sort represent the actual activities. Then, we define events that represent the actions to change activity state and fluents that represent the activity state. An activity state can either be Started or Finished and the events that are responsible for state change are the Start and End events. Then, in the EC model above, we define the Initiates axioms that specify that if the Start event happens at some time, the fluent Started holds true after that time and thus the Initiates axioms represent the state change. As a result of End event, the activity state changes to Finished (represented by the second Initiates axiom). We also define a Terminates axiom, which specifies that as a result of End event, the activity state is no longer Started (and thus the fluent Started does not hold).

We define some axioms to control the invocation of specified events such as the End event should only Happen once the activity has already been started, and the fluent Started Holds. Similarly, other axioms specify that once the activity has Started (or Finished) the Start (and End) events should not Happen. Finally, the last two axioms specify that the initial condition for the fluents that they do not hold at time point 0. In this work, we model different kind of activities that include:
- Activities without intermediate states.
- Activities that may require restart - possibly within loop body.
- Activity model for dynamic task delegation.

Space limitations restrict us to detail the EC models for different kind of activities. However, a detailed discussion can be found in (Zahoor E., 2011). Let us briefly discuss how such an EC based composition model can be used for reasoning by defining two instances (ActivityA and ActivityB) of sort activity. In the EC model shown in Figure 5, we first import the EC core files (root.e and ec.e) and then we import the activity.e file, which contains the EC model for activities modeling shown earlier.

Then, we define instances of sort activity that represent the activities and also the goal for the process that is to have the activities Finished (and thus requiring the fluent Finished to hold) at time-point 2. Finally, we specify the range for time/offset and any options for the DECReasoner, in this case requiring not to show predicates (showpred off).

**Figure 5. Activities definition**

In the model as shown in Figure 4, we first define an EC sort named activity. The instances of the sort represent the actual activities. Then, we define events that represent the actions to change activity state and fluents that represent the activity state. An activity state can either be Started or Finished and the events that are responsible for state change are the Start and End events. Then, in the EC model above, we define the Initiates axioms that specify that if the Start event happens at some time, the fluent Started holds true after that time and thus the Initiates axioms represent the state change. As a result of End event, the activity state changes to Finished (represented by the second Initiates axiom). We also define a Terminates axiom, which specifies that as a result of End event, the activity state is no longer Started (and thus the fluent Started does not hold).

We define some axioms to control the invocation of specified events such as the End event should only Happen once the activity has already been started, and the fluent Started Holds. Similarly, other axioms specify that once the activity has Started (or Finished) the Start (and End) events should not Happen. Finally, the last two axioms specify that the initial condition for the fluents that they do not hold at time point 0. In this work, we model different kind of activities that include:
- Activities without intermediate states.
- Activities that may require restart - possibly within loop body.
- Activity model for dynamic task delegation.

**Web Services.** The proposed composition design also allows modeling the Web services. As the proposed framework aims to reason about the composition process, we only model the core aspects leaving out the details, which are only needed for services execution. However, proposed models can be extended to handle other aspects, if needed. Different Web services models have been proposed, that include:
- Web services with different synchronization modes (synchronous / pull-push based asynchronous).
- Web services that require re-invocation - possibly within loop body.
- Web services to be discovered based on constraints, called Nodes.

The event-calculus model for the synchronous Web services is shown in Figure 6. Space limitations restrict us to detail different EC models and a detailed discussion can be found in (Zahoor E., 2011).
3.3 CONTROL/DATA FLOW SPECIFICATION.

We have proposed the EC based patterns for specifying different control/data flow constructs, such as Dependency, Split/Join and different Split/Join Schemes, Conditional invocation/start of components; Iteration and data flow between different components. Different constructs that are modeled include:

- Dependency, enforcing invocation order.
- Split construct with different split-schemes (parallel/alternative/exclusive).
- Join construct with different join modes (all/exactly-one/at-least-one/subset).
- Conditions and conditional invocation of components.
- Iteration, components re-invocation unless loop exit condition is false.
- Request/Response data associated with and Message flow between components.

The dependency construct specifies the control and/or data flow dependency between different components and requires that the dependent component should not be started/invoked unless the component on which it is dependent, is completed/response data has been received, as shown in Figure 7. In order to model the dependency between different components using event-calculus we can add the axioms to the process specification that specify that the invocation/start event of the dependent component should not happen unless the fluent representing the completion of the source component holds. The EC based generic model for the specification of dependency construct is shown in Figure 8.

\[
\text{Happens(DEPENDENT EVENT, time) } \rightarrow \text{HoldsAt(SOURCE FLUENT, time).}
\]

3.4 MODELING NON-FUNCTIONAL ASPECTS.

The use of EC as modeling formalism also allows modeling different non-functional (temporal and security) aspects. Different patterns for temporal aspects include:

- Response time
- Restart/Refresh
- Invocation time-frame and delay
- Allen’s interval algebra
- Modeling time-units

Here, we only discuss the EC based model to specify the Allen’s temporal relationships. Allen’s Interval Algebra is a calculus for temporal reasoning that was introduced by James F. Allen in 1983. The calculus defines possible relations between time intervals and provides a composition...
table that can be used as a basis for reasoning about temporal descriptions of events.

<table>
<thead>
<tr>
<th>Base relation</th>
<th>Representation</th>
<th>Event-calculus modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP_A before COMP_B</td>
<td>Component A</td>
<td>Component B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specify DELAY between COMP_A completion and COMP_B invocation</td>
</tr>
<tr>
<td>COMP_A meets COMP_B</td>
<td>Component A</td>
<td>Component B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No DELAY between COMP_A completion and COMP_B invocation</td>
</tr>
<tr>
<td>COMP_A overlaps COMP_B</td>
<td>Component A</td>
<td>Component B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specify OVERLAP_DELAY, time in which COMP_A and COMP_B are in concurrent execution</td>
</tr>
</tbody>
</table>

**Figure 10. Base relations for Allen’s Interval Algebra**

The base relations for the Allen’s interval algebra can be mapped and applied to the proposed framework, as shown in Figure 10. We briefly present the patterns for specifying after/meets/overlaps relations using EC in Figure 11. A more detailed discussion about other relationships can be found in (Zahoor E., 2011).

COMP_A after COMP_B

Happens(EVENT_B, time) → Happens(EVENT_A, time + DELAY).
Happens(EVENT_A, time) → Happens(EVENT_B, time - DELAY).

COMP_A meets COMP_B

Happens(EVENT_B, time) → Happens(EVENT_A, time).
Happens(EVENT_A, time) → Happens(EVENT_B, time).

COMP_A overlaps COMP_B

Happens(EVENT_B, time) → Happens(EVENT_A, time - OVERLAP_DELAY).
Happens(EVENT_A, time) → Happens(EVENT_B, time + OVERLAP_DELAY).

**Figure 11. EC patterns for Allen’s interval algebra**

4. PROCESS INSTANTIATION AND VERIFICATION

Once the EC based composition design has been specified, an EC reasoner can be used to instantiate the composition process to find a solution respecting all the functional and non-functional constraints associated with the process. The solution returned by the reasoner states what events happen at which time-points and also shows the effects those events have on the fluents, and the instantiated solution serves as a plan for process execution. The process instantiation phase may result in a number of solutions in the case of loosely constrained process. A particular solution from the set is chosen for execution based on either user-choice or based on some criteria such as overall-cost etc. In reference to temporal properties, one criterion for solution-selection is minimal time requiring to find a solution specifying to complete the execution process in minimal possible time.

As an example, consider the basic EC based model, Figure 5, as shown earlier in the composition design phase. Invoking the EC reasoner for the model gives us the solution shown as shown in Figure 12.

Discrete EC Reasoner 1.0
loading activity instances.e
loading foundations/Root.e
loading foundations/EC.e
loading includes/activity.e
32 variables and 78 clauses relsat solver
1 model
—
model 1:
Happens(Start(ActivityA), 0).
-Started(ActivityA).
+Finished(ActivityA).
+Finished(ActivityB).

**Figure 12. Solution returned by the reasoner**

The solution returned by the reasoner shows that if the Start events happen (representing that the activities are thus being started) at time-point 0, the fluents Start hold at time-point 1 as indicated by the + sign shown next to them at time-point 1 (representing that the activities state has been changed to Started). Once the activities have been started, the End events happen at time-point 1 to have the fluents Finished hold at time-point 2, that was the specified process goal. Note that the End events also make the fluents Started does not hold as indicated by the − sign shown next to them at time-point 2, representing the activity state is no longer Started.

The proposed approach for the verification of declarative Web services composition processes allows for both model checking the verification properties and for identifying and resolving the conflicts in the process specifications a result of process verification. The approach is based on satisfiability solving and requires verification properties to be added to the process specification and then encoding the problem into a SAT problem and then using the SAT-solver to find a solution or a set of unsatisfied clauses. The verification properties are added to the process specification and in general they can either be based on
occurrence (or absence) of some specific event, as represented by EC events, or they can be based on satisfaction (or dissatisfaction) of some property, as represented by EC fluents. Figure 13 shows the general structure of EC axioms for the specification of verification properties.

\[(!)\text{Happens} (\text{SomeEvent()}, \text{SomeTimePoint}) \rightarrow \text{Happens} (\text{SomeOtherEvent()}, \text{SomeTimePoint}).

\[(!)\text{HoldsAt} (\text{SomeFluent()}, \text{SomeTimePoint}).\]

Figure 13. Verification properties specification

The EC to SAT encoding can be very large especially with the increase in time-points/free variables in axioms and with the complexity of the composition process. As a result, the set of unsatisfiable clauses (termed as unsatisfiable core) can be very large. We thus propose to filter the unsatisfiable core and to only consider the encoded clauses of interest. Space limitations restrict us to detail the filtering process, a detailed presentation and example can be found in (Zahoor E., 2011).

5. COMPOSITION MONITORING

Once the composition process is in execution, the composition monitoring stage allows specifying and reasoning about the monitoring properties during process execution. The composition process is specified using the EC and is then used to instantiate, verify and execute the composition process, Figure 14-1. The instantiation phase involves finding a solution to the composition process using the EC reasoner and the instantiated plan is then executed using the execution engine, Figure 14-2.

The proposed monitoring framework, Figure 14-3, works during the composition process execution and is divided into three phases, as discussed below:

5.1 Properties Specification.

The specification phase requires the user to specify the functional and non-functional properties that need 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 14-1) 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, Figure 14-3→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. Furthermore, 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 etc. The choice of highly-expressive EC 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 (Gaaloul et al., 2010) for details). We briefly present EC models for some of these properties in Figure 15.

\[\text{Happens} (\text{StartInvoke}(S1), \text{time1}) \& \text{Happens} (\text{EndInvoke}(S1), \text{time2}) \& \text{time2} - \text{time1} = \text{SomeTimeValue} \rightarrow \text{Happens} (\text{Terminateprocess}(), \text{time2}).

\[\text{Happens} (\text{InvalidateResponse}(\text{service}), \text{time}) \rightarrow \text{Happens} (\text{SendAlertNotification}(), \text{time}).\]

Figure 15: Monitoring properties specification

The first axiom in the model shown in Figure 15 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 → Response and we will later discuss different response actions. The second axiom above specifies to send Alert notification once the response message from any service does not remain valid.

5.2 Detection and Effects Calculation.

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 EC 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 EC fluents. The detection phase may thus require the execution engine support (for instance checking data validity, Figure 14-2).

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, Figure 14-3→1.

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. Adding the partial plan with the violation to the initial plan and re-invoking the reasoner achieve the effects calculation phase. 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 foreseeing 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 EC reasoner to perform deductive reasoning, Figure 14-3→1.

5.3 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 etc) 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 etc. 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 first adding the current process state (with the violation). The reasoner is then re-invoked 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 nonfunctional 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 etc, Figure 14-3→2) and may also require the support from the DECreasoner in order to do abductive reasoning for actions such as finding alternatives, Figure 14-3→1.

6. IMPLEMENTATION DETAILS

The EC models for the proposed framework are specified using the discrete EC language (Mueller, 2006) and all the models mentioned earlier can be directly used for reasoning purposes.

6.1 The ECWS Application.

In order to abstract the EC models from the process-designer and automate the composition process specification, verification and monitoring, we have implemented a Java-based application, called ECWS, that provides a user friendly interface for specifying the composition design. It allows specifying different components and control/data flow between them.

Further, the ECWS application allows to generate EC models for the specification and will automatically invoke the DECreasoner to reason about the generated EC models. The resulting models returned by the DECreasoner are then displayed to the user both in the RAW form and by parsing them and aligning them with a time-modeling approach, Figure 16. The proposed ECWS application for the process specification is still in early phases and only serves as a proof of concept prototype and does not handle process execution and monitoring.
6.2 Enhancements to DECReasoner.

One important limitation of DECReasoner is the time taken for EC to SAT encoding which exponentially increases with the increase in time-points and introducing complex axioms involving multiple free variables, as we discussed in (Zahoor et al., 2010a; Zahoor, Perrin, & Godart, 2010b). 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.

Further, we have thoroughly analyzed and modified the C language code for the encoding process to improve performance. Profiling the encoding process, we identified that the hashing function (DictHash) is not that efficient as it tries to calculate the hash-value based on first 6 characters of the input symbols. However, the structure of input symbols (in general) is such that only last few characters differ from other symbols. This results in a lot of collisions/chaining and subsequent use of strcmp takes all the time. By just changing the hash function to calculate the hash based on last 6 characters of the input symbol, we can avoid a lot of hashing conflicts and this improves performance. The updated hash function is shown below and the changes can be downloaded from DECReasoner official Website.

```c
int DictHash(Dict *d, char *symbol)
{
    unsigned char s[6]; size_t len; len=(size_t)strlen(symbol);
    memset(s,0,6);
    if (len > 6) {
        memcpy(s,symbol+(len-6),6);
    } else {
        memcpy(s,symbol,(size_t)len);
    }
    return (int)(((s[1]+s[5]+(s[0]+s[4])*(long)256) +
                   (s[3]+s[2]*(long)256)*(long)481) % d->size);
}
```

6.3 Performance Evaluation Results.

In this section, we will detail the performance evaluation. The tests were conducted on a MacBook Pro Core 2 Duo 2.53 Ghz and 4GB RAM running Mac OS-X 10.6. The DEC reasoner version 1.0 and the SAT reasoner, relsat-2.0 were used for reasoning. For the performance evaluation results, we consider two case studies. First, we consider a composition process being setup to semi-automate the disaster plan for the Australian National Herbarium (ANH), Canberra, as discussed in (Zahoor E., 2011). We further complicate the example by increasing the number of components (and conditions) and adding the same control/data flow constructs and temporal and security constraints for the newly added components (and conditions).

The time taken for process instantiation (solution computation using relsat solver) and the effect of proposed modifications to the encoding process is shown in Figure 17, with X-axis representing the number of components and the Y-axis representing the time in seconds. The performance results indicate that the modified encoding process (based on the changes we proposed) result in significant performance improvement. The solution computation using the relsat is very efficient.

A detailed discussion about performance evaluation of zchaff solver for the process verification and the effectiveness of the proposed unsatisfiable-core filtering approach can be found in (Zahoor et al., 2012).

7. CONCLUSIONS

In this paper, we have presented the DISC framework, which provides a constraint-based declarative approach for self-healing Web services composition. The proposed DISC framework has three main stages; Composition design, Instantiation and execution, and Composition monitoring.
The composition process starts when the user specifies the composition design using a user-friendly interface allowing him to drag and drop components and provide constraints. The composition design is then used during the instantiation and execution phase to either identify any conflicts (such as deadlocks) or to provide a set of solutions (plans) for the composition process. If the design is conflict-free and a set of solutions is returned during the instantiation, a particular plan (user selected) is then executed and is monitored during the composition monitoring phase. The proposed approach is declarative and advocates the need for a unified formalism to handle different process stages. As a result, it provides recovery actions such as re-planning to find alternatives in case of monitored run-time violations and allows the process to be self-healing. We have also presented implementation details and performance evaluation results.

8. REFERENCES


http://hipore.com/ijsc


Flexible pattern monitoring for WS-BPEL through stateful aspect extension. In IEEE International Conference on Web Services, ICWS'08, pp. 577-584.


Authors

Ehtesham Zahoor did his PhD from Université de Lorraine, France in 2011. He is currently working as an Assistant Professor at National University of Computer and Emerging Sciences, Islamabad, Pakistan. He has been an active researcher in the domain of Web services composition and author of numerous peer reviewed conference/journal publications.

Kashif Munir did his PhD from University of Innsbruck, Austria in 2009. He was a post doctoral researcher at Telecom Bretagne, France from February 2011 to December 2012. He is an Assistant Professor at National University of Computer and Emerging Sciences, Islamabad, Pakistan. He is a reviewer of Elsevier Computer Networks and International Journal of Computer Mathematics and author of numerous peer-reviewed conference and journal publications.

Claude Godart is a Professor at University of Lorraine, France. He has lead and made researches about business process management and Web services composition in complex settings: flexible process models, transactional processes, Web services composition and governance with QoS properties, trusted execution of processes in clouds. He is author and co-author of more than 200 papers, and adviser of around 30 Ph.D. theses. He contributed to several software developments, including the first version of Bonita (http://www.bonitasoft.com/). He is member of the editorial board of several journals (e.g. IEEE TSC, IJSC) and member of many conference program committees (e.g. BPM, ICWS, Cloud, SCC).

Olivier Perrin is full professor at University of Lorraine, France. He has expertise in Business Process Modeling, Web services compositions, workflow management, and enterprise application & data integration. He has published several articles in international journals, and conferences and he was involved in many international and European projects. He has been member and/or chair of many international conferences, and reviewer for IEEE journals.