# DISC: A declarative framework for self-healing Web services composition

Ehtesham Zahoor, Olivier Perrin, Claude Godart

# DISC: A declarative framework for self-healing Web services composition

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*—Web services composition design, verification and monitoring are active and widely studied research directions. Little work however has been done in integrating these related dimensions using a unified formalism. In this paper we propose a declarative event-oriented framework, called DISC, that serves as a unified framework to bridge the gap between the process design, verification and monitoring. Proposed framework allows for a composition design to accommodate various aspects such as data relationships and constraints, Web services dynamic binding, compliance regulations, security or temporal requirements and others. Then, it allows for instantiating, verifying and executing the composition design and for monitoring the process while in execution. The effect of run-time violations can also be calculated and a set of recovery actions can be taken, allowing for the self-healing Web services composition.**

## 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 composite services satisfying user needs and there have been many approaches to model the composition process such as WS-BPEL, WS-CDL and others. However, these approaches tackle the composition problem by focusing on the control flow of the composition using a procedural approach and as pointed out in [1] they over constrain the composition process making it rigid and not able to handle the dynamically changing situations. Further the focus on data, temporal, security aspects and other non-functional requirements is not thoroughly investigated and adding these aspects add a lot to the complexity of the composition process. The design-time verification of the composition process is another important aspect and it aims to identify the conflicts, such as deadlocks, in the composition design before execution. Then, as the Web services provide machine to machine interaction over a network, the monitoring of the composition process when in execution is evident. In a dynamically changing situation (such as a crisis management scenario), Web services are more error prone to failures such as network failures and delays and ideally the composition process should be able to react to these failures and take recovery actions.

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. Specifically we make the following contributions:

**Design that can accomodate various aspects:** The proposed 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 or temporal requirements or other non-functional aspects. Further, it allows for specifying monitoring preferences (called recovery constraints) for the user to specify actions to be taken in case of run-time violations.

**Event calculus for both modeling and monitoring:** Traditionally the web services composition problem is considered as a planning task, given a goal the planner can give a set of plans leading to the goal. However, in case of run-time monitoring we already have a plan to execute and in case of violation it is important to compute the side-effects this violation has on the overall goal. Our approach is based on event calculus and the use of event calculus is twofold, at design ”abduction reasoning” can be used to find a set of plans, and at the execution time ”deduction reasoning” can be used to calculate the effect of run-time violations.

**Extensible approach:** The event calculus allows for integrating the exisiting work on composition design [2], composition monitoring [3], authorization [4], [5], and work on modeling other related aspects. These models thus can be modified and extended and our framework acts as a bridging agent between composition design, verification and monitoring. Further, there have been some approaches that attempt to translate BPEL-based process to event-calculus for verification [6] and also LTL based declarative process to event calculus [7], that justify the expressiveness of event calculus for process specification.

**Composition as a SAT problem:** The proposed implementation framework encodes the event calculus models to boolean satisfiability (SAT) problem for solution finding. This allows to apply the extensive research in the SAT domain for the composition problem. For instance, the proposed verification approach relies on the SAT solver to provide near-miss models and/or unsatisfied clauses. Further,

SAT based problems such as (weighted) MAX-SAT open new directions to explore for services composition.

**Implementation architecture:** The event calculus models are presented using discrete event calculus language [8] and thus can be used directly for reasoning purposes. For process verification, we extended *DECReasoner* [8] to include *zchaff* as a solver and then using *zverify* to find the unsatisfiable core. This also serves as an example of extensibility of the proposed framework as different reasoners can be used to analyze the same SAT-based encoding. Further, we present an implementation architecture addressing how DISC framework can be used in practice.

## II. Motivation and related work

The motivation for our work originates from the need for process modeling, analysis and monitoring in a crisis situation [9]. 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 and most of the proposed approaches for the composition design can be divided into Workflow composition and AI planning based approaches (using situation calculus [10], rule-based planning [11], and others), as discussed in [12]. The problem of traditional Workflow oriented approaches (such as WS-BPEL, WS-CDL or COSMO [13]) 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 [14]). Further 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 [1], [14], [13] 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 a particular automata with guards, and using SPIN model checker [15], BPEL to timed automata and using UPPAAL model checker [16], BPMN constructs into labeled petri-nets then using BPEL2PLML [17] 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 [18], the process algebra based approach to compose and verify the composition process [19] and the restricted abstract BPEL process to analyze the correctness [20] but these approaches are 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 [21], [22], [3]. Current approaches are mostly proposed as a new layer over 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 very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time. In [7] authors attempt to add monitoring directives to a declarative approach but still the approach lacks expressiveness and does not allow for actions such as *replan* and *reinstantiation*. Our work can be compared to [23], 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 does not allow for actions such as replanning and alternative path finding. The DISC framework is an extension of [9], but here, we extend and refine the previous framework to handle SAT-based process verification, present how event calculus can be used for process design, verification and monitoring focusing on the actual event-calculus based models implemented in discrete event calculus reasoner, specify how different reasoning approaches can be used for planning and side-effects calculation and detail an implementation architecture.

## III. Motivating example

Let us consider a sample scenario when the start of the hurricane season and the possible land fall of a category 4 hurricane has prompted the authorities to set up an emergency center to handle any possible injuries. In a typical Web services based setup, the emergency center works by contacting the Web services provided by different systems and a composition process is being thought of to handle the patients. Patient is initially taken to the adjacent initial checkup center and thus the composition process needs to first contact its Web service for patient registration and for fetching the checkup results. Then based on the results received, the composition process may decide to discover and contact some hospital and ambulance Web services (not known in advance) for scheduling the possible surgery and for transportation to selected hospital. This choice will be made using the Web services provided by different

hospitals/ambulance providers and will also be based on certain constraints such as the hospitalization and surgery facilities availability and distance from the initial checkup center. Further the access to the patient information file from the Web service provided by the social security system and contacting the blood bank service for the additional blood supply may also be needed.

In this scenario, it seems unfeasible to specify a strict and pre-defined process due to highly dynamic environment. However the boundaries of any acceptable solution are known and this leads to the choice of declarative approach to model the process. Further, the temporal, security consideration (such as separation of duties) and other non-functional requirements should be taken into account. Then, any solution to the composition process would be based on the design level service contracts and the actual service invocations can be different, due to highly dynamic environment. Thus the process should be self healing and should provide recovery actions to cater for monitoring violations.

## IV. Proposed framework

The proposed DISC framework has three main stages, *Composition design*, *Instantiation and execution* and *Composition monitoring* (see figure 1).

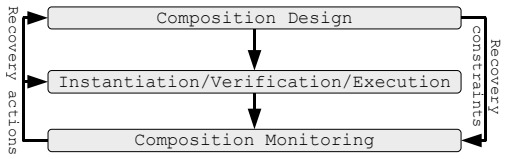


Figure 1. Proposed framework stages

The composition process starts when the user specifies the *composition design*, using a user friendly interface, allowing her/him to drag and drop components and provide constraints. The composition design also includes the conditions to monitor at runtime (called recovery constraints) and the corresponding actions to be taken in case of a violation. This design specified by the user 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. In case of run-time violations, recovery constraints specified during the composition design, are used to find out recovery actions (such as *replan*, *reinstantiate* that allow the process to be self-healing) to be taken. We will now detail the different phases of the DISC framework.

## V. Composition design

### A. Components

The various components that constitute the process design include the concrete Web service instances already known and the *nodes*, that need to be discovered and instantiated based on some constraints. Each node has a unique type such as Hospital, for a detailed discussion about nodes/dynamic binding based on constraints, see [24], [25]. Further, the user can add constraints to the composition design that specify the boundaries for the solution to the composition process. These include the constraints to handle process choreography, nodes binding, and for specifying non-functional properties such as temporal or security constraints. They also include the recovery constraints that specify an activation condition and the corresponding actions to be taken in case of monitoring violations. The possible actions can be *ignore*, *retry*, *reinstantiate* a Web service node by binding another service than the current one, *replan* to find another execution plan based on current situation, and others. Using the actions such as *reinstantiate* or *replan* allows for finding an alternative and thus make the process self-healing as it can learn from run-time failures and propose alternatives. For a more detailed discussion about different constraints see [9].

### B. Event calculus

In order to model the *composition design*, our approach relies on the Event Calculus (EC) [7]. Event Calculus is a logic programming formalism for representing events and their side-effects and can infer "*what is true when*" given "*what happens when*" and "*what actions do*" (see figure 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 [9].

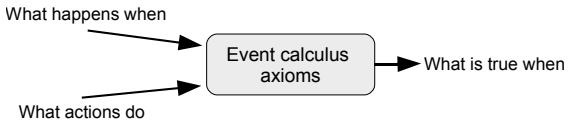


Figure 2. Event calculus components

The EC comprises the following elements: $\mathcal{A}$ is the set of *events* (or actions), $\mathcal{F}$ is the set of fluents (fluents are *reified*[1]), $\mathcal{T}$ 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

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

event calculus predicates used for modeling the proposed framework are:

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

Further, some event calculus axioms are available that relate the various predicates together.

The choice of EC is motivated by several reasons. First, EC integrates an explicit time structure (this is not the case in the situation calculus) independent of any sequence of events (possibly concurrent). Then, given the composition design specified in the EC, a reasoner can be used to instantiate the composition design. Further, EC is very interesting as the same logical representation can be used for verification at both design time (static analysis) and runtime (dynamic analysis and monitoring). Further, it allows for a number of reasoning tasks that can be broadly categorized into deductive, abductive and inductive tasks and 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. This leads to bridging the gap between composition design, verification and monitoring as the same framework is used with different reasoning approaches (see figure 3 for the mapping of event calculus to the proposed framework).

| | |
|---|---|
| *What happens when* | Specified initial orderings (partial plan, if any) and the sought execution plan for the specified goal |
| *What actions do* | Actions such as service invocation, nodes binding and associated effects |
| *What is true when* | The composition design including the Initial situation such as constraints, dependencies and the goal for the composition process (If any). |

Figure 3. EC for the proposed framework

The event calculus models are presented using the discrete event calculus language [8] and 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*) are universally quantified and in case of existential quantification, it is represent with variable name within curly brackets, {variablename}. Further, for spacing issues we will abbreviate *Response* to *Resp*, *Service* to *Serv* and others.

### C. Ground model

At a basic level, the composition process can be regarded as the invocation of the participating Web services and thus the goal of the composition process is to receive the response message from the participating services (for services with request-response invocation mode) and/or to invoke the services (with one-way invocation mode). For sake of simplicity, we will only consider the request-response invocation of services. The EC model below depicts this behavior:

***Ground model** - CM-1.0*

```
sort service, node
fluent RespRecvd(service)      event Invoke(service)
Initiates (Invoke(service), RespRecvd(service), time).
```

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

```
sort service      service Service1, Service2
fluent RespRecvd(service),      event Invoke(service)
Initiates(Invoke(service), RespRecvd(service),time).

!HoldsAt(RespRecvd(service), 0).
HoldsAt(RespRecvd(service), 1) ;composition goal
```

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

```
0     Happens(Invoke(Service1), 0). Happens(Invoke(Service2), 0).
1     +RespRecvd(Service1).      +RespRecvd(Service2).
```

The models presented above are synchronous; however the service invocation can be asynchronous and the composition process can either request and later "pull" the data from provider or alternatively data is "pushed" to the process by service providers, when it is ready. In order to model the pull-based asynchronous invocation, we can update the model *CM-1.0*, and break down the invocation process by adding events and fluents for the sending request and then pulling the response. In order to model the push-based asynchronous invocation, we can introduce the concepts of queues that can be used to store the pushed data from the service providers and composition process can then use the data from the queues.

Further, the composition process may also process transfer of streaming data that can be either service driven or process driven. In order to model the process driven (pull-based) streaming data handling, we introduce the concept of data validity/expiry and introduce an axiom to reinvoke the service once the current data is expired.

***Streaming data - extends CM-1.0***

*fluent RespValid(service)  event InvalidateResp(service)*
*Initiates(Invoke(service), RespValid(service), time).*
*Terminates(InvalidateResp(service),RespValid(service),time).*

*Happens(ReceiveResp(S1),time1)&Happens(InvalidateResp(S1), time2)*
*→ time2 - time1 = 5.*
*!HoldsAt(RespValid(S1),time)→Happens(InvokeServ(S1),time).*

The model above specifies that as soon as the response is received from the service, it is considered valid (first axiom, after fluent/event specification). Then, after a specific amount of time, the data expires (third axiom) and is no longer valid (second axiom) and we further add an axiom that specifies to re-invoke the service in case of data expiry (last axiom).

### *D. Modeling choreography constraints*

Choreography constraints specify the constraints regarding the control flow of the composition process and express the order and execution sequence (partial or complete) of the participating entities. Some examples of choreography constraints include *before*, *after*, *if-then-else*, and *choice*. We will now briefly discuss the dependency constraint, that specifies that a service is dependent on some other service. A detailed discussion on how other choreography constraints (such as split, join,...) can be modeled using event calculus can be found in [2]. Further there have been some approaches that attempt to translate BPEL based process to event calculus for verification [6], that justify the expressiveness of event calculus for process specification as the BPEL constructs can be translated to event calculus.

In order to specify the execution dependency between services, we introduce the *HasDependency* predicate that specifies that the service S1 has dependency on service S2 and thus the service S2 must be invoked/response received before invoking the service S1. The following axiom and predicate added to the model *CM-1.0* handles this behavior:

***Choreography constraints: Dependency - extends CM-1.0***

*predicate HasDependency(S1, S2)*
*HasDependency (service1,service2) & Happens (Invoke(service1), time1) →*
*{time2} HoldsAt (RespRecvd(service2), time2) & time1 >= time2.*

The axiom above specifies that if a service has dependency on some other service, it cannot be invoked before we have received the response from the other service. The model can be extended to add cardinality and other aspects related to process choreography.

### *E. Modeling data flow*

The recent usage and popularity of mashup applications highlights the importance of data handling, as the mashups are data based services composition. The data being received from the Web services may be valid for a specific period, it may be ill-formed or may be in different format than required by the process. While discussing the event calculus models earlier, we intentionally left out the request and response message parameters for the sake of simplicity. Request and response message parameters can be added by creating sorts for request and response messages and correspondingly updating the axioms, this will allow us to define the message or data values dependency. Further, while discussing the streaming services, we specified how data expiry and re-invocation (once data is expired) can be modeled using event calculus and in this section we will briefly discuss the event calculus model for the use of translator data operator, which translates data between different formats.

***Data flow: translator operator - extends CM-1.0***

*sort datatype  fluent Translated(service, datatype)*
*predicate HasType(service, datatype)  event Translate(service, datatype)*
*Initiates(Translate (service, datatype), Translated (service, datatype), time).*

*HasDependency (service1,service2) & HasType (service1, datatype1) & HasType (service2, datatype2) & datatype1 != datatype2 → Happens (Translate (service2, datatype), time).*

In order to model the translator operator, we use the ground model CM-1.0, with the dependency constraints as discussed in the previous section. First we add a new sort, *datatype*, and a predicate *HasType (service, datatype)* that specifies the type of data (XML/JSON) provided by each service. We then introduce an event, *Translate (service, datatype)*, and the corresponding fluent, *Translated (service, datatype)*, that specifies if the data has been translated to some specified format and the related axiom. The core of the model below is the last axiom which specifies that if there is a data dependency between two services and their data types are different then the data provided by service (not having dependency) must be translated to the format accepted by the dependent service.

### *F. Modeling local constraints*

The local constraints are added to Web service nodes for their discovery and binding. To model nodes instantiation, we add the sorts *node* and *constraints* (which specify the constraints added to a node) to the model CM-1.0. The *IsConcrete(service)* fluent separates the concrete Web service instances (used in the composition process) from the services in the repository and candidates for selection. The fluent *Bound(node,service)* specifies if the node has been eventually bound to some service while the fluent *Resolved(node)* specifies that the node has been both bound to some service that has been invoked to get the results.

We also introduce the predicate *HasConst(node, constraint)* which specifies the constraints added to a node and the predicate *SatisfiesConst(service, constraint)* that specifies the constraints satisfied by the service. The predicate *HasType(service, node)* specifies the type of each service and we add some events that use the fluents discussed above. We update the service invocation axiom presented in model CM-1.0 to only handle the invocation for the concrete Web services (and not to invoke the services in repository unless they are bound to some Nodes and are made concrete).

***Local constraints - extends CM-1.0***

*sort node, constraint*
*fluent IsConcrete(service), Bound(node,service), Resolved(node)*
*predicate SatisfiesConst(service,constraint), HasConst(node,constraint), HasType(service,node)*

*event Resolve(node), Bind(node, service)*
*Happens(Invoke(service), time) → HoldsAt(IsConcrete(service), time).*

Next, we add axioms to handle node binding. These axioms satisfy that a service is bound to a node only if it satisfies constraints and has the same type as of the node. This binding results in service being marked concrete (and thus can be invoked), finally once the service is bound to node and is invoked the node is considered resolved.

***Local constraints - Axioms***

*Initiates(Bind(node, service), Bound(node, service), time).*
*Initiates(Bind(node, service), IsConcrete(service), time).*

*HasConst(node, constraint) & !SatisfiesConst(service, constraint) → !Happens(Bind(node, service),time).*
*Happens(Bind(node1, service),time) & HasType(service, node2) → node1 = node2.*

*Initiates(Resolve(node), Resolved(node), time).*
*Happens(Resolve(node),time) → {service} HoldsAt(Bound(node, service), time) & HoldsAt(RespRecvd(service), time).*

The basic approach for handling nodes instantiation using the event calculus discussed above, requires transforming the service descriptions from service repository into event calculus predicates and fluents, it does incur some overhead. As a result, moving the local constraints specification and discovery outside event calculus (see [24] for a SQWRL based approach that searches through the OWL-S based repository using SQWRL queries) may be a possible option. In this approach, the nodes are discovered and the candidate services for a node are added to event calculus with the axioms that exactly one service is executed (see section-V-H for an example).

### G. Non-functional and Recovery constraints

Using event calculus as the modeling framework allows for specifying the non-functional constraints such as temporal and security constraints. We have already discussed the data expiry constraints, that specify the temporal constraint on data validity. We can also specify the temporal constraints on the complete or partial composition process by specifying the execution time between services. Further, the actions with delayed effects can be modeled by breaking the invocation process into two actions that mark the start and end of the action. As an example, to model the time taken by a service we can break down the *Invoke(service)* into *StartInvoke(service)* and *EndInvoke(service)*. Regarding the security constraints it is possible to model the authorization policies and other aspects as proposed in [4], [5]. As an example, the model below specifies the separation of duties constraints for two services S1 and S2.

*Happens(Invoke(S1), time1) & time2 > time1 → !Happens(Invoke(S2), time2).*

For modeling the recovery constraints our approach is based on Event Processing Network model as we proposed in [9]. Recovery constraints take the form of axioms with an *activation condition* part and an *action* part. As an example, the recovery constraint to terminate the execution in the case of response time delay for service S1, takes the following form:

*Happens(StartInvoke(S1), time1) & !HoldsAt(RespRecvd(S1), time2) & time2 - time1 = 10 → Happens(Terminate(), time2).*

The above axiom comprises an activation condition and the corresponding action and can be regarded as *activation condition → action*. The action *Happens (Terminate(), time2)* specifies to terminate the execution and there are other possible actions: *ignore*, *retry*, *reinstantiate* (bind another service than the current service), *replan* (find another execution plan based on current situation). Using the actions such as *reinstantiate* or *replan* allows for finding an alternative and thus make the process self-healing as it can learn from run-time failures and propose alternatives.

### H. Example

We now review the motivating example and discuss how the *composition design* can be specified using our model. We consider that the service *InitCheckup* takes 10 minutess while the services *SomeAmb* and *SomeOtherAmb* take 5 and 8 minutes respectively (assuming the process needs a confirmation once the patient reaches the hospital). These services are pull-based asynchronous and are modeled using *StartInvoke* event (which models request) and *EndInvoke* event (which marks the eventual *pull* for the response). Further we assume that the Hospital and Ambulance nodes have already been resolved (using external approach based on (see [24] as we discussed earlier) and the possible candidates (*SomeHosp, SomeOtherAmb...*) are added to the event calculus model. In the model below, we first define the instances of the basic sorts, the Web services.

*service InitCheckup, SocSecurity, SomeHosp, SomeAmb, SomeOtherAmb...*
*Happens(StartInvoke (InitCheckup), time1) & Happens(EndInvoke (InitCheckup), time2) → time2 -time1 = 10.*
*Happens(Invoke(SocSecurity), time1) & Happens(InvalidateResp(SocSecurity), time2) → time2 -time1 = 20.*
*Happens (StartInvoke (InitCheckup), time1) & !HoldsAt (RespRecvd(InitCheckup), time2) & time2 -time1 != 10 → Happens(ReInstantiate(Ambulance), time+3).*

In the model above the first axiom specifies the response time for *InitCheckup* service (we have omitted the same for other services). The second axiom specifies the data expiry for the *SocSecurity* service to be 20 minutes and in the last axiom we add a recovery constraint to *ReInstantiate* in the case of response time delay for the *InitCheckup* service. For binding the Ambulance node the initial constraints include to prefer (if possible) the road-service over the air-service as there are limited air-ambulances. Then, in the model below, we add the initial situation for the fluents that they does not hold at time point 0 and the dependencies that exist between services. Then, as our proposal aims to select one

user selected Web service as a result of nodes instantiation, we add some axioms to handle this behavior.

*!HoldsAt(ResponseRequested(service),0)...*
*HasDependency(BloodBank, SocSecurity) ...*
*Happens(Invoke(SomeHosp), time1) & time2 > time1 → !Happens(Invoke(SomeOtherHosp), time2).*

We use the axioms from the previous section to model different aspects such as dependencies resolution and others. Finally we specify the goal, which is to get the response from the selected ambulance/hospital service at time point 20, *HoldsAt(RespRecvd(SomeAmb),20) | HoldsAt(RespRecvd(SomeOtherAmb),20).*

## VI. Instantiation, Verification and Execution

The EC model for the composition design is then used to instantiate, verify and execute the composition process. The instantiation phase involves both binding the nodes to the concrete Web service instances and finding a solution to the composition process using the event calculus reasoner. The various concepts related to the nodes instantiation such as service compatibility rules, propagation and backtracking are detailed in ([24]). The process instantiation phase attempts to find a solution (plan) to the composition process respecting the associated constraints. A plan is a sequence of EC *Happens* clauses that specify the temporal ordering of different actions, whose execution leads to the goal. As our proposal allows for specifying only the boundaries to the composition process and as with the case of nodes instantiation phase, the instantiation may result in a number of solutions and the user can be given option to choose one solution and the chosen plan is then executed.

If there are some conflicts in the composition design and/or the specified constraints are too strict, this leads to empty solution set and requires the verification of the composition design to identify any conflicts or hard constraints. Our approach to verification relies on the SAT solver to provide a set of near-miss models and/or unsatisfied clauses. As an example consider the temporal constraint added to the composition process, saying that the services S1 and S2 should not execute concurrently. In case of planning, the reasoner will only generate the solutions that will respect this constraint, but if no such solution exists and the only solution is to execute them concurrently to achieve the goal, the planner can return a near-miss model highlighting the strict constraint.

Delegation of verification task to the SAT solver has many benefits. First, in relation to the proposed implementation framework, the *DECReasoner* [8] attempts to find a solution by transforming the EC model into a SAT problem and invoking SAT solver for the solution, thus the same SAT encoding can be used for verification purposes. Then, it provides an highly extensible approach, same SAT encoding can be either analyzed by multiple solvers. Further, it allows not only for the conflicts (such as deadlocks) detection, but allows for identifying the hard constraints that should be relaxed to find a solution and for identifying other side-effects such as the data expiry and others. In reference to the motivating example, introducing a cyclic dependency (*BloodBank* dependending on *SocSecurity* and vice-versa) leading to deadlock and then invoking the reasoner (*zchaff* and then *zverify_df*) gives us a set of unsatisfied clauses including the following:

*1634 0 - HasDependency(BloodBank, SocSecurity)*
*1640 0 - HasDependency(SocSecurity, BloodBank).*

In reference to motivating example, using the EC reasoner for the composition design specified earlier gives a set of solutions (plans), including the one mentioned below:

```
0   Happens(StartInvoke(InitCheckup), 0).
1   +InvocationStarted(InitCheckup).
2   Happens(Invoke(SocSecurity), 2).
3   +RespRecvd(SocSecurity). +RespValid(SocSecurity).
    Happens(Invoke(BloodBank), 3).
4    +RespRecvd(BloodBank). +RespValid(BloodBank).
    ...
10  Happens(EndInvoke(InitCheckup), 10).
11  +RespRecvd(InitCheckup). +RespValid(InitCheckup).
    Happens(Invoke(SomeOtherHosp), 11). Happens(
    StartInvoke(SomeOtherAmb), 11).
12  +InvocationStarted(SomeOtherAmb). +RespRecvd(SomeOtherHosp).
    +RespValid(SomeOtherHosp).
    ...
19  Happens(EndInvoke(SomeOtherAmb), 19).
20  +RespRecvd(SomeOtherAmb). +RespValid(SomeOtherAmb).

1 predicates, 0 functions, 3 fluents, 4 events, 45 axioms
encoding 2.0s solution 0.4s total 2.9s
```

## VII. Composition monitoring

The composition monitoring process is divided into three phases. The *detection* phase is responsible for detecting the monitoring violation and this phase maintains 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. The *side-effects calculation* phase is responsible to deduce the side-effects of the detected monitoring violation. This is handled by using the deductive reasoning by adding the partial plan (with violation) and re-invoking the reasoner. Finally the *recovery* stage is responsible for using the user preferences for recovery action, specified as recovery constraints, to cater for and recover from the violation. As discussed earlier, using the actions such as *reinstantiate* or *replan* allows for finding an alternative and makes the process self-healing as it can learn from run-time failures and propose alternatives. In case there is no recovery constraint available, a set of possible actions can be provided to choose from.

Let us now review the motivating example and see how composition monitoring works by first discussing how the *event repository* is populated and then the mismatch detection and recovery (self healing). In reference to the initial instantiated plan, let us consider that *InitialCheckup* Web

service is invoked at time-point 0, followed by *SocSecurity* Web service at time-point 2 (according to instantiated plan from previous section) and then the response is received. We add the following messages exchanges to the repository:

*Happens(Invoke(InitialCheckup), 0) ...*
*Happens(Invoke(SocSecurity), 2).*
*HoldsAt(RespRecvd(SocSecurity), 3). HoldsAt(RespValid(SocSecurity), 3).*

The initial instantiated plan shows that the response from the service *InitialCheckup* is then received at time-point 10, however unlike the design level contract, the service takes 12 minutes instead of 10 (specified in the design level contract) and the event repository is updated as below:

*Happens(EndInvoke(InitCheckup), 12).*
*HoldsAt(RespValid(InitCheckup), 13). HoldsAt(RespValid(InitCheckup), 13).*

This in-turn results in a mismatch between the initial instantiated plan (see previous section) and the actual execution sequence and to calculate the side-effects caused by this violation, we add the current execution sequence to the composition design model (removing the goal and design level contracts for the services leading to a deductive reasoning task) and re-invoke the reasoner. We get the following model:

*21* *Happens(EndInvoke(SomeOtherAmb), 21).*
*22* *+RespRecvd(SomeOtherAmb). +RespValid(SomeOtherAmb).*
*Happens(InvalidateResp(SocSecurity), 22).*
*23* *-RespValid(SocSecurity).*

The above model shows that we both miss the goal and the response from the *SocSecurity* service does not remain valid. For binding the Ambulance node, initially one user selected service was chosen (possibly the road-ambulance service as there are limited air-ambulances) from the candidate services. However, as the action associated with the recovery constraint is to *reinstantiate* the ambulance node in case of delay, this will allow the process to be self-healing as now another candidate service can be chosen (possibly air-ambulance) to meet the goal. To handle reinstantiation, we do abductive reasoning to find another execution plan by retaining the goal, adding the current execution sequence to the composition design and re-invoking the model:

*14* *Happens(Invoke(SomeHosp), 14). Happens(StartInvoke(SomeAmb), 14).*
*...*
*19* *Happens(EndInvoke(SomeAmb), 19).*
*20* *+RespRecvd(SomeAmb). +RespValid(SomeAmb).).*

The above model shows that as a result of reinstantiating, we get an updated model suggesting to use the *SomeAmb* instead of *SomeOtherAmb* to avoid missing the specified goal.

## VIII. Implementation Architecture

In order to test our proposal, we have implemented the proposed model using the discrete event calculus language [8] and all the models mentioned earlier can be used for reasoning purposes.

The composition process starts when the user specifies the composition design, using a user friendly interface as similar

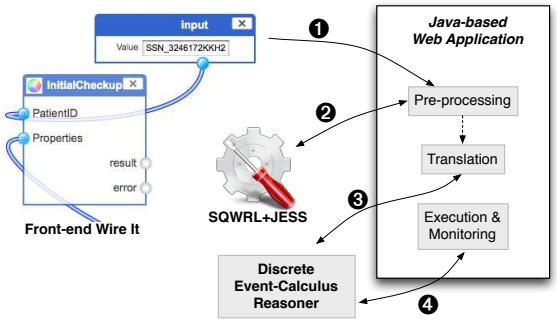


Figure 4. Implementation architecture

to *Yahoo! Pipes* implemented using the *WireIt* Javascript library, allowing to drag and drop components and provide constraints (see figure 4-1). Then, the Java-based application translates the composition design to event calculus based model in three phases. The *pre-processing* phase discovers and binds the Web service nodes to concrete Web services instances using the SQWRL based approach [24] (see figure 4-2). The nodes instantiation process itself can be purely event-calculus based as we discussed earlier, but due to performance issues we propose to use the pre-processing phase. Then, the *selection* phase allows user to select one particular service from the candidate services for the node binding. The *translation* phase follows which does the event calculus translation using the following guidelines. For all the services, instances of the sort service are created. Then the dependencies between the services are translated into dependency predicates, the temporal constraints are added and an event calculus file is generated. The EC reasoner is then invoked to process the file to either provide possible solutions or detect conflicts (see figure 4-3). The result is then used by the Java application to perform the actual services execution, if a solution is found and selected by the user (see figure 4-4). The semantics of events (such as invoke) can be roughly mapped to BPEL for execution and while the process is in execution, an event repository data structure is maintained at the Java application layer. This repository is updated with every service call and response reception and is compared to the initial solution returned by the reasoner at each step for the process monitoring. In case of a violation/recovery actions, event calculus translated file is updated and sent to reasoner.

In order the evaluate the performance of the proposed framework, we have tested the event-calculus model for the motivating example using the DEC reasoner. 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. The motivating example requires 2 seconds for encoding the problem into event calculus and 0.4 seconds

for solution finding during the instantiation phase and to test scalability of the approach, multiplying the problem ten times (resulting a composition problem with 70 Web services) the process takes 16.4 seconds for encoding and 0.7 seconds for solution finding. In general the encoding process does not scale well especially with the increase in time-points and as a future work, we aim to modify the proposed DECreasoner encoding [8] to make the process faster. The verification phase uses *zchaff/zverify* solver and takes 0.4 second for 70 services. For the process reinstantiation, a key observation is that it always takes less time than the initial solution as we do have a partial plan and that reduces the problem. Space limitations restrict us to detail the evaluation results further.

## IX. Conclusion

In this paper we present the DISC framework, which provides a constraint based declarative approach for self-healing Web services composition. Our approach allows user to design the composition by identifying the participating entities and by providing a set of constraints that mark the boundary of the solution. The obtained design is backed up by an EC based model which allows for specifying many different aspects such as partial or complete process choreography and exceptions, data relationships and constraints, dynamic binding, compliance regulations, security requirements and others.

The design is then instantiated/verified using the EC reasoner and the resulting (user selected) plan is chosen for execution. The monitoring phase follows and the effect of run-time violations on the process execution can be calculated and a set of recovery actions (such as terminate, reinstantiate, replan) can be taken. We have presented an example crisis management scenario that highlights our approach and have briefly discussed the implementation architecture.

## References


[1] 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.

[2] N. K. Cicekli and Y. Yildirim, "Formalizing workflows using the event calculus," in *DEXA*, 2000.

[3] K. Mahbub and G. Spanoudakis, "A framework for requirents monitoring of service based systems," in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA: ACM, 2004.

[4] A. K. Bandara, E. C. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," *Policies for Distributed Systems and Networks, IEEE International Workshop on*, vol. 0, p. 26, 2003.

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

[6] W. Fdhila, M. Rouached, and C. Godart, "Communications semantics for wsbpel processes," in *ICWS*, 2008, pp. 185–194.

[7] R. A. Kowalski and M. J. Sergot, "A logic-based calculus of events," *New Generation Comput.*, vol. 4, no. 1, pp. 67–95, 1986.

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

[9] E. Zahoor, O. Perrin, and C. Godart, "An integrated declarative approach to web services composition and monitoring," in *WISE*, 2009, pp. 247–260.

[10] S. A. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," in *KR*, 2002.

[11] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid, "Composing web services on the semantic web," *VLDB J.*, vol. 12, no. 4, 2003.

[12] J. Rao and X. Su, "A survey of automated web service composition methods," in *SWSWPC*, 2004.

[13] D. A. C. Quartel, M. W. A. Steen, S. Pokraev, and M. van Sinderen, "Cosmo: A conceptual framework for service modelling and refinement," *Information Systems Frontiers*, vol. 9, no. 2-3, pp. 225–244, 2007.

[14] M. Pesic and W. M. P. van der Aalst, "A declarative approach for flexible business processes management," in *Business Process Management Workshops*, 2006.

[15] X. Fu, T. Bultan, and J. Su, "Analysis of interacting bpel web services," in *WWW*, 2004, pp. 621–630.

[16] N. Guermouche and C. Godart, "Timed model checking based approach for web services analysis," in *ICWS*, 2009, pp. 213–221.

[17] R. Dijkman, M. Dumas, and C. Ouyang, "Formal semantics and analysis of bpmn process models," Technical Report Preprint 7115, Queensland University of Technology, Tech. Rep., 2007.

[18] X. Yi and K. Kochut, "A cp-nets-based design and verification framework for web services composition," in *ICWS*, 2004, pp. 756–760.

[19] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and reasoning on web services using process algebra," in *ICWS*, 2004, pp. 43–.

[20] Z. Duan, A. J. Bernstein, P. M. Lewis, and S. Lu, "A model for abstract process specification, verification and composition," in *ICSOC*, 2004, pp. 232–241.

[21] 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.

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

[23] 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.

[24] E. Zahoor, O. Perrin, and C. Godart, "Rule-based semi automatic web services composition," in *SERVICES I*, 2009, pp. 805–812.

[25] C. Pautasso and G. Alonso, "Flexible binding for reusable composition of web services," in *Software Composition*, 2005, pp. 151–166.