Web services governance : Security and Data handling aspects

Ehtesham Zahoor

To cite this version:

Ehtesham Zahoor. Web services governance : Security and Data handling aspects. Web. Université Nancy II, 2011. English. NNT : . tel-00643552

HAL Id: tel-00643552
https://theses.hal.science/tel-00643552

Submitted on 22 Nov 2011

Université Nancy 2

ÉCOLE DOCTORALE IAEM

Département de formation doctorale en informatique

THÈSE

présentée et soutenue publiquement le

pour l'obtension du

Doctorat de l'Université Nancy2 (spécialité informatique)

par

Ehtesham ZAHOOR

Gouvernance de service: aspects sécurité et données

Thèse dirigée par Claude GODART et Olivier PERRIN

préparée au LORIA, Projet SCORE

Jury :

Rapporteurs :			
	Ernesto DAMIANI	-	Professeur à Université de Milan
	Farouk TOUMANI	-	Professeur à Université Blaise Pascal
Examinateur :	Jörg HOFFMANN	-	Dr à INRIA Nancy Grand Est
	Yves LE TRAON	-	Professeur à Université du Luxembourg

Laboratoire Lorrain de Recherche en Informatique et ses applications – UMR7503

Acknowledgments

The work presented in this thesis would not have been possible without the help of my thesis advisors Claude Godart and Olivier Perrin. I am sincerely and highly thankful to them for being highly supportive and for their continuous guidance and wisdom throughout this journey.

I am very fortunate to have had opportunity to be part of the team SCORE at LORIA and I would like to thank my colleagues Karim Dahman, Walid Fdhila, Aymen Baouab and others for all the time we spent together during these past years. I would also like to thank my parents, my friends Atif and Dawood, and my wife for their prayers and support.

Abstract

Web services are in the mainstream of information technology, paving way for inter and across organizational application integration. Individual fine-grained services may need to be orchestrated into more coarse-grained value added processes and Web services composition is highly active and widely studied research direction. In the literature, a number of approaches have been proposed to handle different aspects at different stages of composition process life-cycle, that includes composition process design, verification and execution time monitoring. The traditional approaches however focus only on some stages of process life-cycle and little work however has been done in integrating these related dimensions using a unified formalism.

At the process design stage, the proposed modeling approaches are mostly procedural and they over-constrain the process making it rigid and assuming the design choices that may not be present in the requirements but only added to specify the process flow. For the process verification, the traditional approaches require to map the process to some formal logic and then verify the process. However, this lack of integration results in a complex model and it may not always possible to have a complete transformation from one modeling approach to other. Further, with the addition of non-functional (such as security and temporal) requirements the transformation becomes even more complex and challenging. Then, the traditional approaches for composition process monitoring while in execution, build upon composition frameworks that are highly procedural, such as *WS-BPEL*, an this in-turn poses two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not part of the composition framework and functional and non-functional properties are not expressed in terms of events and their effects. Secondly the use of procedural approach for process specification does not bridge the gap between organization and situation in a way that it is very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time, and it does not allow for a reasoning approach allowing for effects calculation and recovery actions such as re-planning or alternate path finding.

The objective of our thesis is thus to handle the process modeling, design-time verification, execution-time monitoring and recovery in an **integrated** and **declarative** way. Declarative approach results in a highly flexible composition process that may be needed to cater for dynamically changing situations while integration simplifies the approach by using the same formalism for composition design, verification and monitoring. The use of declarative and integrated approach further allows to have recovery actions such as re-planning (to cater for monitored violations during process execution) which are difficult to achieve using traditional approaches. In this thesis, we have proposed an integrated 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.

The proposed framework allows for a composition design that is declarative and 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. We have based the composition design on event-calculus and defined patterns for specifying the functional and non-functional aspects using event-calculus for process specification, instead on relying on different formalisms or extensions for specifying different aspects as required by traditional approaches such as *WS-BPEL*.

For the process design-time verification we have proposed a symbolic model checking approach using satisfiability reasoning. The need for the satisfiability solving for process verification stems from the fact the state space of a declarative process can be significantly large, as the process is only partially defined and all the transitions may not have been explicitly defined (in contrast to procedural approaches), and thus it makes it easier to do the symbolic model checking instead of using explicit representation of state transition graphs and/or using the binary decision diagrams. Further, as the conflict clauses returned by the SAT solver can be very large, we have proposed filtering criteria to reduce the clauses and defined patterns for identifying the nature of conflicts.

For the execution-time monitoring (and recovery from any monitored violations) we have proposed an event-based message-level monitoring approach that allows to reason about the events and does not require to define and extract events from process specification, as the events are first class objects of both design and monitoring frameworks. As the proposed monitoring approach builds upon event-calculus based composition design, it allows for the specification of monitoring properties that are based on both functional and non-functional (such as temporal, security or their combinations) requirements. These properties are expressed as event-calculus axioms and can be added to the process specification both during process design and during the process execution. The proposed monitoring approach both allows for KPI's measurement (that may be needed for process evaluation or result in proactive detection of any violations) and the detection of violations once they happen. Different levels of detection are provided such as detection to the process execution plan, detection to the violations based-on any properties and events added during process execution and others.

Contents

Part I

PROLOGUE

Chapter 1

Introduction

Contents

Web services are in the mainstream of information technology, paving way for inter and across organizational application integration. They can be defined as the software systems designed to support interoperable machine-to-machine interaction over a network[1]. Their definition highlights one of their core objectives, interoperable machine-to-machine interaction. Thus, the Web services allow heterogenous systems based on heterogenous platforms to not only communicate but to expose their operation to the rest of the world using Web services. In addition, the Web services can also be used to implement reusable application-components, such as currency conversion, weather reports and others.

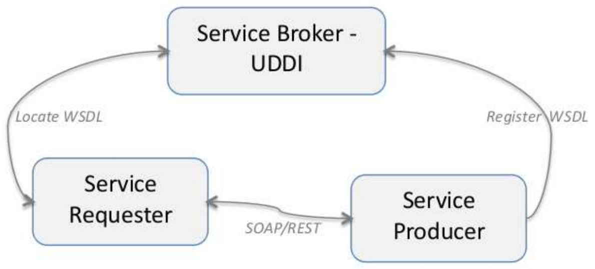


Figure 1.1: Basic elements of a Web service

Web Services have three basic elements (Figure-1.1): their description specified using the Web Services Description Language *(WSDL)* which is a W3C standard and an XML-based language for locating and describing Web services, a directory

[1]http://www.w3.org/TR/ws-gloss/

service where companies can register and search for Web services (or more specifically their interfaces described by *WSDL*) called Universal Description, Discovery and Integration *(UDDI)* and a communication protocol to access the services which can be either a standardized XML-based communication protocol called *(SOAP)* or Representational state transfer *(REST)*.

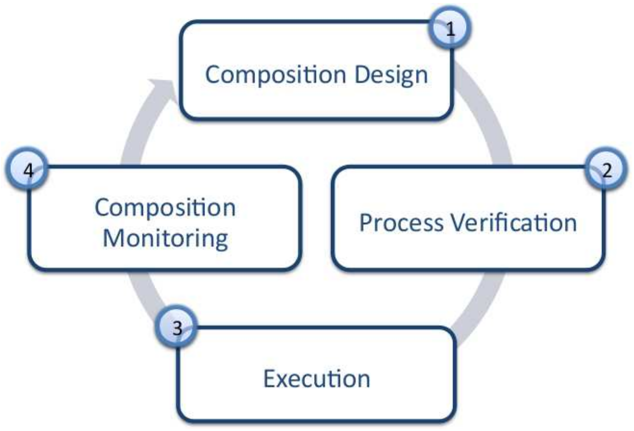


Figure 1.2: Web services composition process life-cycle

The Web services are autonomous and they only present an interface that allows other systems to use the operations provided by them and internal implementation of these operations is hidden to the outside world. Further, 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 (Figure-1.2-1). Then the composition process needs to be verified to identify any anomalies and conflicts (such as deadlocks) in the process specification (Figure-1.2-2) and once the process has been verified it is executed (Figure-1.2-3). Then, as the Web services are autonomous and only expose their interfaces, composition process is based on design level service contracts and the actual execution of composition process may result in the violation of the design-level services contracts due to errors such as network or service failures, change in implementation or other unforeseen situation. This highlights the need to monitor and detect the errors and react accordingly to cater for them. The reaction may also require to update the composition design to cater for the monitored violation and to find alternatives (Figure-1.2-4).

1.1 Motivation

The motivation of our work stems from the process modeling, design-time verification, execution-time monitoring and recovery in an integrated and declarative way to cater for dynamically changing situations (for instance, crisis handling or the logistics processes). The situations these processes are dealing with are complex, ambiguous, highly dynamic and characterized by temporal and security constraints, coordination of multiple services and multiple data sources, with varying degrees of reliability and in different formats. Information that was treated at time t may be superseded by new information at time t+1. The process to handle such a situation thus needs to be highly flexible, may only be partially defined and is required to be self-healing and self-adaptive to handle continuously changing situation.

Web services composition is highly active and widely studied 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 (not all) stages of process life-cycle and this lack of integration results in a complex model such as mapping the *WS-BPEL* based process specification to a particular automata with guards, and using SPIN model checker [Fu 2004] for verification, *WS-BPEL* to timed automata and using *UPPAAL* model checker [Guermouche 2009b] for checking temporal properties. Further, the lack of integration leads to the approaches that does not allow (unless with complex mappings) to learn from the run-time failures to provide the recovery actions, such as re-planning or alternative path finding, to recover from the monitored run-time violations based on current state of the process. Then, the second shortcoming of proposed approaches is that they aim to build on top of the traditional approaches (such as *WS-BPEL*, *OWL-S*) which focus on the control flow of the composition using a procedural approach and as pointed out in [van der Aalst 2006] 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 and other non-functional requirements is not thoroughly investigated and adding these aspects makes the composition even more complex, as again they are proposed as a new layer on the top of existing *WS-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.

1.2 Thesis objectives and main contributions

The objective of our thesis is to investigate a non-procedural approach for Web services composition that integrates different stages of the process life-cycle in an **unified** and **declarative** way. Declarative approach results in a highly flexible composition process that may be needed to cater for dynamically changing situations while integration simplifies the approach by using the similar formalism for composition design, verification and monitoring. The use of declarative and integrated approach further allows to have recovery actions such as re-planning (to cater for monitored violations during process execution) which are difficult to achieve using traditional approaches.

In this thesis, we have proposed an integrated 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. Below we briefly discuss our main contributions.

1.2.1 Declarative composition design

The proposed framework allows for a composition design that is declarative and 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. We have based the composition design on event-calculus and defined patterns for specifying the functional and non-functional aspects using event-calculus for process specification, instead on relying on different formalisms or extensions for specifying different aspects as required by traditional approaches such as WS-BPEL.

The use of event-calculus as the modeling formalism allows for integrating the existing work on composition design [Cicekli 2000], composition monitoring [Mahbub 2004], authorization [Bandara 2003, Gaaloul 2010], and work on modeling other related aspects. The proposed models in the thesis thus can be further 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 *WS-BPEL* based process to event-calculus for verification [Fdhila 2008] and also LTL based declarative process to event calculus [Kowalski 1986a], that justify the expressiveness of event calculus for process specification.

1.2.2 SAT-based process verification

For the process design-time verification we have proposed a symbolic model checking approach using satisfiability reasoning. The need for the satisfiability solving for process verification stems from multiple sources. First, as the composition process may be declarative and partially defined by only specifying the constraints that mark the boundary of the solution to the composition process and the objective is to find solution(s) that respect those constraints (and which is connectivity verification property), satisfiability solving can thus be used to solve the problem by encoding it as a satisfiability problem, representing the constraints.

Further, the state space of a declarative process can be significantly large, as the process is only partially defined and all the transitions may not have been explicitly defined (in contrast to procedural approaches), and thus it makes it easier to do the symbolic model checking instead of using explicit representation of state transition graphs and/or using the binary decision diagrams. The verification properties can include the connectivity, compatibility and behavioral correctness (safety and liveness properties) and the proposed approach allows for both model checking the verification properties and for identifying and resolving the conflicts in the process specifications a result of process verification. Further, as the conflict clauses returned by the SAT solver can be very large, we have proposed filtering criteria to reduce the clauses and defined patterns for identifying the nature of conflicts.

1.2.3 Event-based process monitoring

For the execution-time monitoring (and recovery from any monitored violations) we have proposed an event-based message-level monitoring approach that allows to reason about the events and does not require to define and extract events from process specification, as the events are first class objects of both design and monitoring framework. As the proposed monitoring approach builds upon event-calculus based composition design, it allows for the specification of monitoring properties that are based on both functional and non-functional (such as temporal, security or their combinations) requirements. These properties are expressed as event-calculus axioms and can be added to the process specification both during process design and during the process execution. The proposed monitoring approach both allows for KPI's measurement[2] (that may be needed for process evaluation or result in proactive detection of any violations) and the detection of violations once they happen. Different levels of detection are provided such as detection to the process execution plan, detection to the violations based on any properties and events added during process execution and others.

Further, the web services composition problem is traditionally 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 process execution. 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. This also allows to cater for the "ripple effect" any violation has on the process execution, and for proactive detection of any possible violation that is bound to happen later in the process execution, as a result of current detected violation. In addition, once a violation is detected and a recovery solution

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

(for instance re-planning) is sought, the proposed approach allows both to find a new solution based on the current process state (thus specifying what steps should be taken now to recover from the violation and hence termed forward recovery) or to backtrack to some previous activity (if possible) and try to find a new from there. Then, any recovery solution takes care of the functional and non-functional properties associated with the process, when performing recovery.

1.2.4 Implementation architecture

The proposed event-calculus models presented in this work are mentioned using the discrete event calculus language [Mueller 2006] and they can be directly used for reasoning purposes. We have proposed an implementation architecture and implemented a Java-based application that allows to abstract the event-calculus models to the end-user. It facilitates the composition design process by providing a user-friendly interface for specifying composition design (including entities and control/data flow between them) and allows to automatically generate the corresponding event-calculus models, invokes the reasoner and shows the results returned by the reasoner.

The proposed approach uses the *DECReasoner* as the event-calculus reasoner, however as we discussed in [Zahoor 2010a, Zahoor 2010b] the event-calculus to SAT encoding process provided by the reasoner, does not scale well. We have thus modified the DECReasoner code to gain substantial performance improvement as evident in performance evaluation results (Section-9.4). Further, we have presented a real world crisis management case-study and discussed how a process-based approach can be beneficial. For process verification, we extended *DECReasoner* [Mueller 2006] 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.

1.3 Thesis Structure

The rest of the thesis is organized as follows: first we will briefly discuss the context and background needed to understand the problem domain, explicitly formalize the problem and present the motivating example in Chapter-2. Further in Chapter-3, we will discuss the related work, focusing on the traditional approaches proposed in the literature deal for composition process design, verification and monitoring. We will also provide a detailed synthesis highlighting the limitations these approaches have in terms of being procedural, lack of expressivity and lack of integration.

Then, in Chapter-4 we will discuss the proposed event-calculus based composition design by first presenting the event-calculus models for different components that constitute the composition design. We will then discuss the event-calculus based models for specifying different control/data flow constructs in Chapter-5 and will then extend the composition process design in Chapter-6, for the incorporation of non-functional (such as temporal and security) requirements. We will organize the models in a pattern-based approach and we will specify the generic patterns for specifying different functional and non-functional properties and this will allow to re-use the patterns for the process specification.

Then we will discuss the proposed symbolic model checking approach using satisfiability reasoning for composition process verification, Chapter-7, and the event-based monitoring (and recovery) in Chapter-8. We will then discuss the implementation details in Chapter-9 presenting the architecture, the *ECWS* tool, the changes to the *DECReasoner* encoding process to improve performance and detailed performance evaluation results based on the motivation example. Finally we will discuss the limitations of the proposed approach, future perspectives to conclude the thesis in Chapter-10.

Part II

BACKGROUND

Chapter 2

Context and problem definition

Contents

2.1 Context

In this section we will briefly discuss the different concepts and background knowledge needed for understanding the problem domain. We will first discuss how the Web services can be defined using the Web Services Description Language *(WSDL)* and how they can be accessed using the *(SOAP)* communication protocol. Further, we will briefly discuss the Service-Oriented Architecture *(SOA)*. Then, we will discuss the Business Process Modeling Notation *(BPMN)* for defining and the Business Process Execution Language *(WS-BPEL)* for implementing the orchestration of Web services into added value processes. Further, we will also briefly discuss the background needed and motivation for using the event-calculus as the modeling formalism.

2.1.1 Web Services Description Language (WSDL)

Web Services Description Language (*WSDL*) is used to describe a Web service and it specifies the location of the service and the operations the service exposes to the rest of the world. The service description using *WSDL* takes the form of an XML document and *WSDL* uses following major elements: *<types>* specify the data types while the *<message>* element specify the messages used by the web service. The *<portType>* element is used to specify the operations a Web service provided while the *<binding>* element specifies the communication protocols used by the service. A skeleton WSDL document[1] for service description is shown below:

Structure of a WSDL document

<definitions>

<types> definition of data types used by the Web service. </types>
<message> definition of messages used by the Web service. </message>
<portType> definition of a port type supported by the Web service. </portType>
<binding> definition of a service binding. </binding>

</definitions>

2.1.2 The *SOAP* communication protocol

SOAP is a XML-based standardized platform and language independent communication protocol that allows for communication between applications over the internet. *SOAP* was designed to allows communication between applications over HTTP, instead of Remote Procedure Calls (RPC) between objects. A *SOAP* message is a XML document containing the following elements: an *Envelope* element that identifies the XML document as a *SOAP* message, a *Header* element that contains header information, a *Body* element that contains call and response information and a *Fault* element containing errors and status information. A Skeleton *SOAP* message[2] is shown below:

Skeleton SOAP Message

<?xml version=1.0?>

<soap:Envelope>
The SOAP Envelope element is the root element of a SOAP message and defines the XML document as a SOAP message.

[1]http://www.w3schools.com/wsdl/wsdl_intro.asp
[2]http://www.w3schools.com/wsdl/wsdl_intro.asp

<soap:Header>
The optional SOAP Header element (which should be first child of envelope element, if present) contains application-specific information (like authentication etc.) about the SOAP message.
</soap:Header>

<soap:Body>
The required SOAP Body element contains the actual SOAP message intended for the ultimate endpoint of the message.

<soap:Fault>
The optional SOAP Fault element is used to indicate error messages and can only appear once in a SOAP message.

</soap:Fault>
</soap:Body>
</soap:Envelope>

2.1.3 Service-Oriented Architecture (SOA)

Web services can also be used to implement a Service Oriented Architecture (*SOA*) which is defined to be a flexible set of design principles for system development, that rely on the use of services to support the development of distributed, interoperable and agile applications[3]. The vision associated with the *SOA* is to separate functions into services, which are autonomous, platform independent and are accessible over a network. As the Web services are autonomous, independent of platforms and programming languages and are accessible over Internet, they can make functional building-blocks of *SOA*.

Further, the SOA allows users to combine and reuse the services to create new applications and for the Web services, high-level languages such as *WS-BPEL* and specifications such as *WS-CDL* and *WS-Coordination* can be used for the orchestration of fine-grained services into more coarse-grained business services. Before going into details of *WS-BPEL* for Web services composition process orchestration, we will briefly discuss the Business Process Modeling Notation (*BPMN*) that can be used to model the composition process at an abstract level and the *BPMN* diagrams can then be converted into *WS-BPEL* for their execution.

2.1.4 Business Process Modeling Notation (BPMN)

The Business Process Modeling Notation (*BPMN*) is a graphical notation for specifying business processes based on a flowcharting technique. The BPMN is intended to be a standard notation that is understandable by business analysts who create the processes, the technical developers responsible for implementing the processes, and the business managers who monitor and manage the processes. Process modeling using *BPMN* is simple and intuitive and it has a small set of of graphical elements that are categorized into following categories[4]:

[3]http://en.wikipedia.org/wiki/Service-oriented_architecture
[4]http://en.wikipedia.org/wiki/Business_Process_Model_and_Notation

- Flow Objects including Events (that specify something that happens), Activities (that specify the kind of work which must be done) and Gateways (that represent forking and merging of paths).
- Connecting Objects which include Sequence Flow (which shows in which order the activities will be performed), Message Flow (that represent what messages flow) and Association (which is used to associate an Artifact to flow objects).
- Swimlanes which include Pool (which represents participants in a process) and Lanes (that are used to organize and categorize activities within a pool according to function or role).
- Artifacts including Data Object (representing input/output data of an activity), Group (to group different activities) and Annotations.

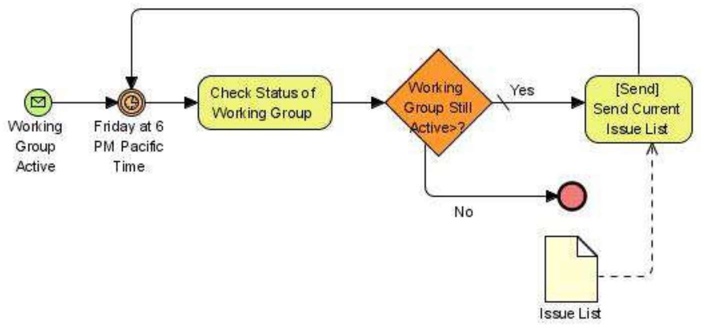

Working Group Active
Friday at 6 PM Pacific Time
Check Status of Working Group
Working Group Still Active>?
Yes
[Send] Send Current Issue List
No
Issue List

Figure 2.1: A simple BPMN process from the BPMN specification

A simple BPMN process with two activities, taken from the *BPMN* specification, is shown in Figure-2.1. As the primary objective of *BPMN* includes to bridge the gap between different stake-holders including business analysts and developers responsible for implementing the process, the BPMN specification includes an informal and partial mapping from *BPMN* to *WS-BPEL*, a more detailed mapping has been implemented in a number of tools such as *BPMN2BPEL*. However, any such mapping is partial and it is possible that some *BPMN* models cannot be translated into *WS-BPEL*. Further, the problem of synchronization of *BPMN* diagrams and *WS-BPEL* so that any modification to one is propagated to the other, is even more complicated.

2.1.5 Business Process Execution Language (WS-BPEL)

Web Services Business Process Execution Language (*WS-BPEL*) is an XML based programming language to specify the orchestration of fine-grained services into more coarse-grained value added processes. As the *WS-BPEL* is an XML-based programming language, it has three basic components: Programming logic that is defined using *WS-BPEL* itself, Data types from XSD (XML Schema Definition) and Input/Output specified using the *WSDL*. A sample *WS-BPEL* process is shown below:

Skeleton WS-BPEL Message

```
<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/" ... >

<import importType="http://schemas.xmlsoap.org/wsdl/" location="" ... />

<partnerLinks>
<partnerLink name="..." partnerLinkType="..." partnerRole="..."/>
</partnerLinks>

<variables>
<variable ... />
</variables>

<invoke partnerLink="..." operation="..." inputVariable="..." />
</process>
```

2.1.6 Event-calculus

In order to model the *composition design*, our approach relies on the Event Calculus (EC) [Kowalski 1986a]. 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 2.2). 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.

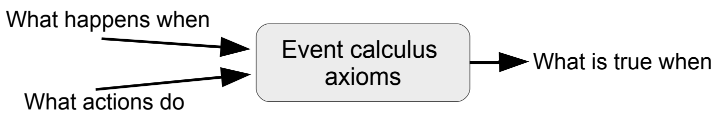


Figure 2.2: Event calculus components

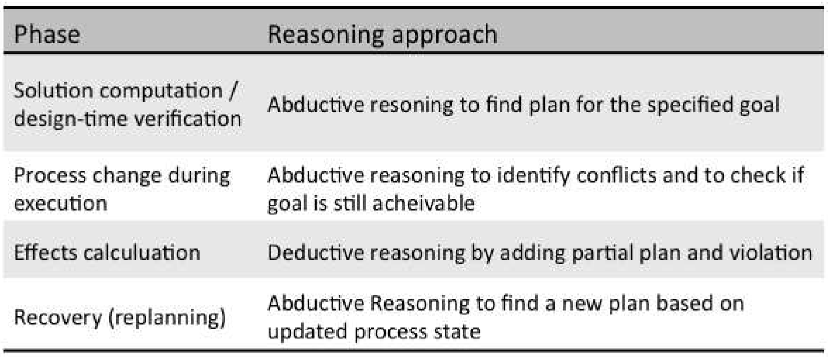
The EC comprises the following elements: $\mathcal{A}$ is the set of *events* (or actions), $\mathcal{F}$ is the set of fluents (fluents are *reified*[5]), $\mathcal{T}$ is the set of time points, and $\mathcal{X}$ is a set of objects related to the particular context. In EC, events are the core concept that triggers changes to the world. A fluent is anything whose value is subject to change over time. EC uses predicates to specify actions and their effects. Basic event-calculus predicates used for modeling the proposed framework are:

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

[5]Fluents can be quantified over and can appear as the arguments to predicates.

- $Happens(e, t)$ is true iff event e happens at timepoint t.
- $HoldsAt(f, t)$ is true iff fluent f holds at timepoint t.
- $Initially(f)$ - fluent f holds from time 0.
- $Clipped(t_1, f, t_2)$ - fluent f was terminated during time interval $[t1, t2]$.
- $Declipped(t1, f, t2)$ - fluent f was initiated during time interval $[t1, t2]$.

The choice of using EC as the modeling formalism 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. 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 2.3 for the mapping of event-calculus to the proposed framework). In addition, the use of event-calculus results in an extensible approach as the proposed models can be modified and extended in order to model other related aspects.



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

Figure 2.3: Event-calculus for the proposed framework

The event calculus models presented in this thesis are modeled using the discrete event-calculus language [Mueller 2006]. All the variables (such as *service*, *time*) 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 some words and the abbreviated words would be used in models and any such abbreviation would be explicitly mentioned before presenting the model.

Further, we will use the *DECReasoner*[6] as the event-calculus reasoner. The *DECReasoner* is a program for performing automated commonsense reasoning using the discrete event calculus. It supports different reasoning modes (as required by the proposed approach) including deduction, abduction, model finding and others. It also supports the commonsense law of inertia, conditional effects of events, release from the commonsense law of inertia, indirect effects of events, state constraints, causal constraints and other aspects allowing to model real-world domains. In order to use the *DECReasoner*, domain description (specified using Discrete Event Calculus Reasoner language) that includes an axiomatization describing domains of interest, observations of world properties at various times, and a narrative of known event occurrences is placed in a file. The *DECReasoner* is then invoked for the domain description and it firsts transforms the domain description into a satisfiability (SAT) problem. It then invokes a SAT solver, which produces zero or more solutions and resulting solutions are decoded and displayed to the user.

2.2 Motivating example

Before formalizing the problem statement, let us now briefly discuss the motivating example that we will use as a base for describing various aspects of the proposed approach discussed in this thesis.

2.2.1 Overview

For the motivating example, we consider a composition process being setup to semi-automate the disaster plan for the Australian National Herbarium (ANH), Canberra[7]. The choice of the motivating example stems from multiple objectives. First, it highlights the need for a process based approach to respond to a crisis situation, which is normally defined informally in the document form. A formal approach will allow to thus verify the recovery process to identify any possible conflicts and anomalies in process specification. Further, the composition process to handle such a situation is highly dynamic and possibly partially defined, is characterized by temporal constraints, coordination of multiple services and multiple data sources, and require the composition process to be self-healing and flexible to adapt to continuously changing environment.

The possible threats to the ANH collection include threats from fire and water and it is suggested that the risks due to bush-fire and lightening strikes are very real due to its location at the base of Black Mountain and close proximity to bushland. The immediate response to handle any fire-based disaster situation (for simplicity, we will only consider the fire-based disaster situation) includes evacuating the building in response to the fire alarm and it is suggested that re-entry of the building(s) and recovery may not commence until the ACT Fire Brigade and

[6]http://decreasoner.sourceforge.net/
[7]http://www.anbg.gov.au/cpbr/disaster-plan/

any other Emergency Service that may be in attendance have given permission. Further, the recovery should not commence until after a plan has been developed for conducting the recovery. The disaster plan suggests a role-based approach and different user groups have been assigned different roles for disaster handling including the *Collections Recovery Coordinator (CRR)*, the *Facilities Coordinator (FC)* and the *Salvage Controllers (SC)*.

The role of collection recovery coordinator (CRR) include the overall management of recovery process such as ensuring security, lighting, and delegating the task of notifying staff to assist in the recovery operation, staff (and any volunteers) management and logging of hours worked. Further the disaster plan suggests that the CRR should nominate someone to deal with media enquiries. In addition, the CRR should ask someone to contact the Bureau of Meteorology to obtain a updated local forecast which may determine the course of action to be taken (continuing storms may require SES to rig tarpaulins over the damaged area). CRR should also monitor progress and adjust plans (to cater for bottlenecks, complaints from salvage workers, the effect of the environmental conditions and others) as appropriate. The communication with insurance company and progress update to senior management is also required. Then, the task of facilities coordinator (FC) is to ensure that the recovery Teams have the proper equipment needed to clean, salvage and stabilize the damaged collection materials. Further, the Salvage Controller (SC) is responsible for initial cleanup, salvaging, sorting and stabilization of damaged collection materials by creating recovery teams.

Before going further, let us briefly review the tasks assigned to CRR and if a process-based approach can be beneficial. First, a lot of tasks have been assigned to CRR and the tasks of assigning shifts and recording the hours worked by staff and volunteers would be very difficult to handle manually. Then it is suggested that someone should notify the staff, someone should contact meteorology department and someone should handle media inquires, how that someone is chosen is left to the choice of CRR. In contrast, using a process-based approach the tasks for allocating user shifts can be automated, a service-based approach to handling media inquires can be provided, meteorology department Web service can be contacted for getting the local forecast and re-adjusting the plans and the task of notifying users can be automated. Further the insurance department Web service can also be contacted.

2.2.2 Case Study: Recovery of priority items at ANH

For this case-study we consider an unfortunate fire-incident as a result of lightning strike and the recovery of the items for priority salvage and treatment at ANH. These items include the *Type specimens* and as a result of fire, they may get smoke-damaged and the recovery plan suggest to either freeze or transfer them to some other location. The recovery process however cannot begin, unless the fire has been (partially) contained and emergency staff is given permission to enter the premises and a plan for recovery has been developed by the CRR in collaboration with FC and SC. Now we will briefly describe the process by first highlighting different entities that constitute the composition process and then the control/data flow, temporal and security constraints for the process. Different entities for the motivating example include:

Web services: The composition process can invoke different **Web services** such as once the fire-alarms are activated, the composition process contacts the Web services provided by different emergency services such as fire-brigade (*FireBrigadeWS*), police (*PoliceWS*), ambulance (*AmbulanceWS*) and also invokes a service to call emergency handling staff (*CallEmergencyStaff*). Further, the meteorology department (*MetDepartmentWS*) and insurance company Web service (*InsuranceWS*) can also be invoked by the composition process, instead of relying on CRR to designate someone to contact them. Further, it is important to contact the meteorology department Web service as soon as possible to get the timely updates needed for planning.

Conditions: Conditional invocation of different Web services (or activities) is modeled using **conditions** such as the arrival of disaster handling staff (CRR/FC/SC) is modeled as conditions *Has(CRR/FC/SC)Arrived*. The help from a professional conservator to provide advice on-site to help the SC and recovery teams in sorting material may be needed. We consider the service to call emergency staff (*CallEmergencyStaff*) also contacts the conservator and his arrival is modeled by the condition, *HasConservatorArrived*. The decision to seek external help (if needed) is modeled as condition *IsExternalHelpNeeded*). Further, the fire-alarm resulting in composition process invocation, can also be only a false-alarm and it is modeled as a condition called *NotAFalseAlarm*.

Activities: Different activities represent the task carried out by the recovery teams such as at the arrival of fire brigade staff, the fire containment starts which is modeled as an activity *FireContainment*). Then the task to examine the nature and extent of damage is modeled as an activity called *ExamineDisasterSite*, the task to plan the recovery is modeled as the activity *PlanRecovery*. The FC/SC collaborate for the recovery of priority items, and the recovery task is modeled by an activity called, *RecoverPriorityItems*.

The process is however likely to change, with new activities can be added once the process is in execution. We now specify the control/data flow for the composition process by identifying dependencies that exists between different entities discussed above. We will also present the composition process modeling, using the flow chart like general notation.

- The fire-alarm can also be only a false-alarm , so if the CRR is available at site

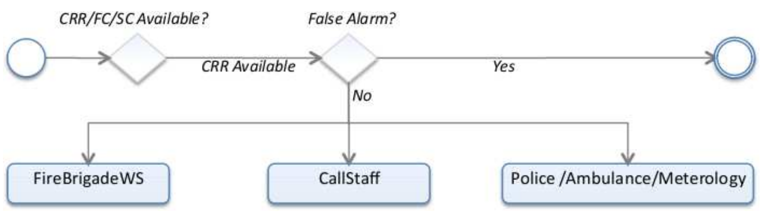


Figure 2.4: Motivating example - CRR available at site

(modeled as a condition, *IsCRRAvailable*), he can verify if it is not the case (modeled as a condition *NotAFalseAlarm*) and invoke the emergency services if needed, Figure-2.4.

- However, if the CRR is not available, the composition process can invoke the services *CallEmergencyStaff* and *FireBrigadeWS* itself, and once the fire-brigade reaches site (*HasFireBrigadeArrived* is true), they can verify if it is not just a false alarm (*NotAFalseAlarm*). The *FireContainment* activity can only start if the fire-brigade staff has decided that it is not just a false alarm (*NotAFalseAlarm*). Further, once the CRR reaches the site he can decide to invoke other emergency services than *CallEmergencyStaff* and *FireBrigadeWS*, if needed. The composition process model from Figure-2.4 can thus be updated as shown in 2.5.

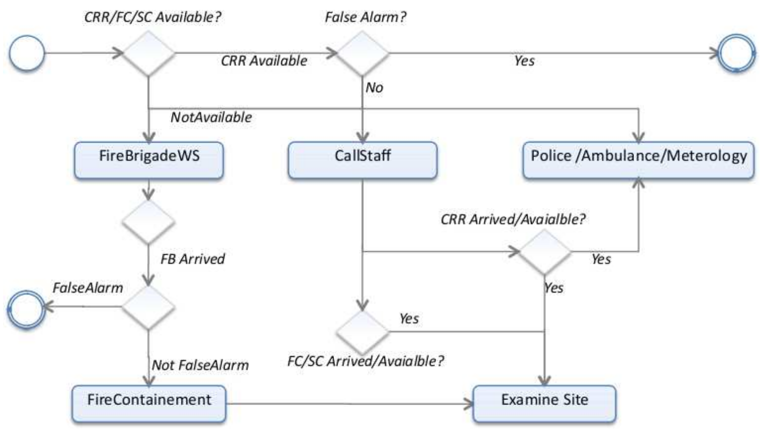


Figure 2.5: Motivating example - CRR not available at site

- Activity for examining the disaster site (*ExamineDisasterSite*) cannot be started unless the CRR has reached the site and *FireContainment* activity is finished. Then, for deciding on the recovery Plan (*PlanRecovery* activity) and deciding on if external help is needed (*IsExternalHelpNeeded*) , CRR, FC and SC should have reached the site. However *PlanRecovery* activity can be started once the CRR is there (and others may not have yet reached the site) but to finish the *planning* process, approval from all three is needed. Further, the priority items recovery (*RecoverPriorityItems*) activity cannot be started unless the *FireContainment* activity has finished and recovery has been planned (*PlanRecovery* activity finishes). The updated composition model is shown in Figure-2.6.

Let us now briefly discuss different temporal and security constraints associated with different services and activities in the composition process, and the constraints associated with the overall goal for the composition process. These temporal and security constraints are only tentative and are likely to change during process execution, for instance the time needed for fire-containment cannot be determined in advance and is dependent on nature and extent of fire.

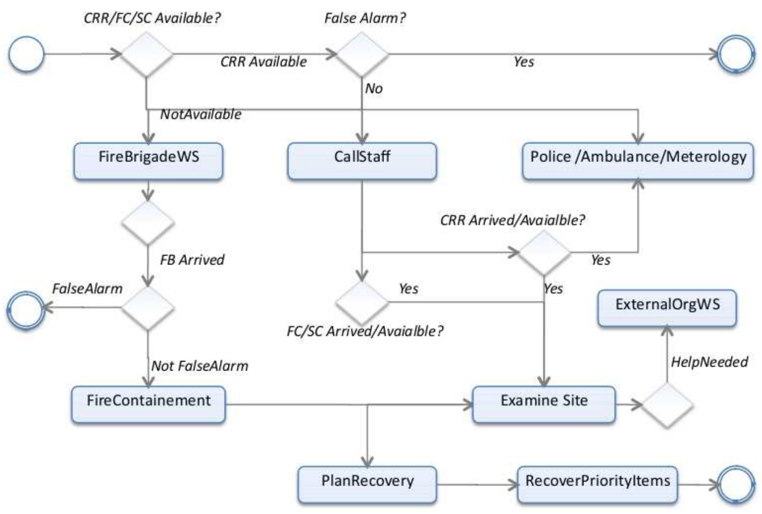


Figure 2.6: Motivating example - a procedural approach

- The smoke damaged, wet specimens should be treated as soon as possible and any delay of more than an hour may result in deterioration of the specimens. This serves as the composition process goal to ideally have the RecoverPriorityItems activity finished before one hour.
- It is estimated that the time needed for ambulance, fire-brigade and police to reach the site can be probably 15 minutes. However, these are only estimated delays and the arrival time for emergency services can vary based on factors such as weather, time and others.
- The arrival time for the staff (CRR/FC/SC/Conservator) varies, however it is estimated that it may take between 15-25 minutes for the staff to reach the site.
- Only once the fire-brigade reaches the site, the time required for (possibly partial) fire-containment can be somewhat estimated.
- The emergency Web services (*FireBrigadeWS*, *PoliceWS* and others) should only be invoked by the CRR, however if the CRR is not available (emergency

situation in non-working hours), the process automatically invokes *FireBrigadeWS* and *CallEmergencyStaffWS* and wait for CRR arrival for invoking other services.

2.3 Problem definition

In this section we will provide an overview of the problem being solved in this thesis and leave the detailed synthesis for limitations of traditional approaches, for Chapter-3. We will discuss the problem by first discussing the lack of integration and limitations of traditional approaches at each composition process life-cycle stage and then will discuss the motivating example highlighting the challenges it poses.

2.3.1 Lack of integration

Web services composition is highly active and widely studied research direction and in the literature a number of approaches have been proposed to handle different aspect related to the composition process at different stages of composition process life-cycle. The traditional approaches focus only on some stages of process life-cycle and this lack of integration results in a complex model such as mapping the *WS-BPEL* based process specification to a particular automata with guards, and using SPIN model checker [Fu 2004] for verification, *WS-BPEL* to timed automata and using UPPAAL model checker [Guermouche 2009b] for checking temporal properties. Further, it is not always possible to have a complete transformation from one modeling approach to other and with the addition of non-functional (such as security and temporal) requirements the transformation becomes even more complex and challenging. In addition, the lack of integration leads to the approaches that does not allow to learn from the run-time failures to provide the recovery actions, such as re-planning or alternative path finding.

2.3.2 Procedural composition model

A process model is called procedural when it contains explicit and complete information about the flow of the process but only implicitly keeps track of why these design choices have been made and if they are indeed part of the requirements or merely assumed for specifying the process flow [Goedertier 2008]. Although specifying exact and complete flow adds a lot to the control over the composition process, however as there is tradeoff between the control and flexibility, this control comes at the expense of process flexibility and thus making the process rigid to adapt to continuously changing situations and possibly not even conforming to the process specification requirements. On the other hand, a process is termed as declarative when it models the minimal (and only specified) requirements that mark the boundary of the process and any suitable execution plan which meets the specified requirements is sought.

The proposed graph-based composition modeling approaches are mostly procedural and although the graph-based approaches tend to be simpler and intuitive for the process modeler for the composition design, they over-constrain the process assuming the design choices that may not be present in the requirements but only added to specify the process flow. The paradigm change from procedural to declarative process modeling was advocated in [Pesic 2006] by introducing the ConDec language for declarative process modeling. However, we believe that the traditional AI planning (and so as rule-based) approaches for composition process modeling are declarative and they have advantages in terms of expressibility, flexibility and adaptability and dynamism as they are more expressive based and are based on formal logic and flexible as only constraints that mark the boundary of the process are specified. Further, the declarative approaches allow for specifying more workflow patterns than the procedural ones.

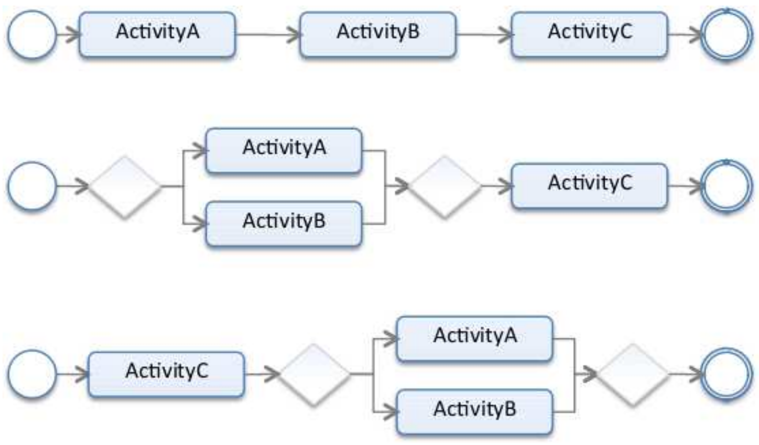


Figure 2.7:

As an example let us consider a simple process with three activities *ActivityA*, *ActivityB* and *ActivityC*. The only requirement regarding the control flow for the composition process is that the all the activities should not be started in parallel. A procedural approach will require enumerating all the possible combinations such as the few shown in the Figure-2.7. In contrast a declarative approach will only specify the constraint (such as *if ActivityA has started then neither ActivityB nor ActivityC can start at the same time*) which will serve as a boundary for any solution to the composition process, without explicitly enforcing a solution.

2.3.3 Verification

The proposed approaches for the composition verification, in general, require mapping the process (mostly defined using procedural approaches such as *WS-BPEL*) to some formal logic (such as petri-nets, automata or process logic) and then using model checkers to verify the composition process. This transformation based approach has two major limitations, first the proposed verification approaches are based on traditional procedural approaches and as we will detail in the Chapter-3, procedural approaches have less expressibility, flexibility and adaptability and dynamism as compared to the declarative ones. Further, the limited expressibility makes it difficult to verify the non-functional properties (such as temporal, security requirements or more importantly their combinations) associated with the composition process as for first, it is difficult to specify the non-functional properties using the traditional approaches such as *WS-BPEL* and a number of approaches have been proposed as an extension to *WS-BPEL* for specifying non-functional aspects and then it is not trivial (if possible) to add formal semantics to them for their verification.

Specifying the exact and complete sequence of activities to be performed for the composition process, as required by the traditional procedural approaches, however does make it possible to use proposed automata or Petri-nets based approaches for design-time verification of composition process. In contrast, for the declarative approaches the process may be only partially defined and thus this makes it difficult to use traditional approaches for the process verification as the transition system for a declarative process can be very large. Further, the design-time verification should be coupled with execution-time monitoring and complexity of these approaches make them difficult to use for verifying the functional and non-functional constraints associated with the process while handle process change or recovery.

2.3.4 Event-based monitoring

Traditional approaches for the composition monitoring are proposed as an extension to some particular run-time and are tightly coupled and limited to it. In contrast the use of an event-based approach works on the message-level and thus is unobtrusive, independent of run-time and allows for integration of other systems and processes, as discussed in [Moser 2010]. Then, the traditional monitoring approaches [Barbon 2006a, Baresi 2009, Mahbub 2004] build upon composition frameworks that are highly procedural, such as *WS-BPEL*, an this in-turn poses two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not part of the composition framework and functional and non-functional properties are not expressed in terms of events and their effects. Secondly the use of procedural approach for process specification does not bridge the gap between organization and situation in a way that it is very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time, and it does not allow for a reasoning approach allowing for effects calculation and recovery actions such as re-planning or alternate path finding as we discussed in [Zahoor 2010a].

2.3.5 Synthesis for the motivating example

Let us briefly review the motivating example and discuss the limitations and challenges the traditional approaches pose while modeling such a composition process.

We first consider the composition model specification that is traditionally specified using the workflow based approaches such as BPMN and later mapped to *WS-BPEL* for process execution. Figure-2.6 shows an attempt to model the composition process using a general flow chart based notation, focusing on only entities and the control flow between them. As evident from the model, the composition process is very complicated and the semantics are not clear leading to many interpretation for the single process. Adding temporal, security and other constraints will further complicate the model and with the increase of entities and their interactions, it would be difficult and time consuming even to have a proper visual representation for the process model.

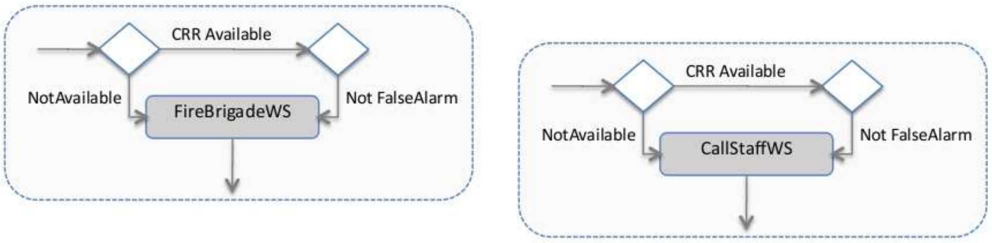


Figure 2.8: Motivating example - a declarative approach

In contrast the proposed declarative approach allows user to specify the constraints (including control/data flow, security, temporal and others) from an individual service or activity perspective (see Figure-2.8). This allows to specify the composition process in a flexible way without over-constraining the process and only specifying the constraints that are part of the problem (and not added for merely process modeling). These constraints mark the boundary of any acceptable solution and a reasoner can then be used to connect these process fragments, respecting the associated constraints.

Further, the successful execution of the process is challenging provided highly dynamic and continuously changing situation. For instance, the time-taken by the fire-brigade and emergency staff to reach the site can be somewhat defined and estimated but the time-taken for the fire-containment process itself is highly relative. Then, it may not be possible to define all the monitoring properties during composition design and even if the design-level constraints are respected, the occurrence of external events during process execution can have impacts on the process execution.

2.4 Summary

In this chapter, we have first briefly discussed different concepts and background knowledge needed for understanding the problem domain in Section-2.1. We have discussed how the Web services can be defined using the Web Services Description Language *(WSDL)* and how they can be accessed using the Simple Object Access Protocol *(SOAP)*. Further, we will discuss the Service-Oriented Architecture *(SOA)*.

Then, we will discuss the Business Process Modeling Notation *(BPMN)* for defining and the Business Process Execution Language *(WS-BPEL)* for implementing the orchestration of Web services into added value processes. Further, we have also briefly discussed the background needed and motivation for using the event-calculus as the modeling formalism.

Then we have presented the motivating example in Section-2.2, that we will use as a base for describing various aspects of the proposed approach discussed in this thesis. For the motivating example, we have considered a composition process being setup to semi-automate the disaster plan for the Australian National Herbarium (ANH), Canberra. We have presented the motivation for choosing the disaster-plan as the motivating example and expressed the example in detail highlighting different components, control/data flow and security and temporal constraints amongst the components.

Further, in Section-2.3 we have have formally presented the problem that is being considered in this thesis and identified limitations of proposed approaches in the literature in terms of having lack of integration and limitations and rigidness of having procedural composition model. We have also discussed the challenges and motivation for design time verification and event-based execution time monitoring of declarative process and how the lack of integration makes it difficult to have recovery actions such as re-planning once a violation is detected during process execution.

Chapter 3

State of the art

Contents

The event-based approach for Web services composition presented in this thesis both aims to integrate different stages of process life-cycle, Figure-1.2, and advocates the use of declarative event-based approach at different stages. In this chapter, we will therefore briefly discuss the proposed approaches in the literature for handling each (or some) of process life cycle stages.

3.1 Composition process modeling

The composition process modeling is the first and most important stage of the composition process life cycle, Figure-3.1. The objective of composition process modeling is to provide high-level specification independent from its implementation that should be easily understandable by process modeler who create the process, the developers responsible for implementing the process, and the business managers who monitor and manage the process.

In the literature, different approaches for the composition process modeling have been proposed having their strengths and weaknesses in terms of ease of usage

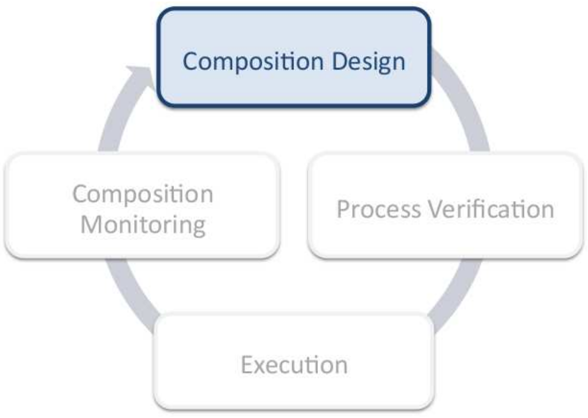


Figure 3.1: Web services composition process design

and expressibility of the modeling language. Further, as it is important to have a process model properly defined, verified, and refined before being implemented, different approaches offer varying level of complexity for model-checking the composition process. The proposed approaches for the composition process modeling can be broadly categorized into Workflow based composition and AI planning based approaches, as discussed in [Rao 2004a] or similarly as either graph-based or rule-based approaches as discussed in [Lu 2007]. In this section, we will first briefly discuss the proposed graph (workflow) and AI planning based approaches and then the proposed approaches for incorporating non-functional (security and temporal) requirements. Then, we will provide a detailed synthesis highlighting the challenges posed by and the short-comings of the traditional approaches for modeling services composition.

3.1.1 Graph based modeling approaches

The graph (or workflow) based process modeling approaches allow for the specification of process definition in an intuitive and easy way using graphical process models based on graph theory or variants. Activities within the process are represented as nodes and control flow and data dependencies between activities is defined using transitions or arcs between activities. However, graph-based process models can include arbitrariness and lack strictness allowing for different interpretation of same process models (or multiple models for the same process requirements). As a result, many graph-based models have their origin in Petri net theory (or on different variants of Petri nets to provide extra expressibility and functionality) including High Level Petri nets [Ellis 1993], Low Level Petri nets [Wikarski 1996], and Colored Petri nets [Merz 1994], a detailed discussion can be found in [Janssens 2000]. Petri nets based approaches have the advantage of providing the formal semantics despite the graphical nature of the process and that in-turn allows for analyzing and verifying the properties such as deadlocks.

The proposed graph-based process modeling approaches includes the Business Process Modeling Notation (BPMN) which provides a graphical notation for specifying business processes using a flowcharting technique very similar to activity diagrams from Unified Modeling Language (UML). BPMN aims to serve as a standard notation that is understandable by all business stakeholders and allows to bridge the gap between business process design and implementation. The proposed modeling approaches also includes [Sadiq 1999] in which authors propose to divide the workflow modeling into structure, data, execution, temporal, and transactional partitions for workflow modeling. The authors have also proposed the FlowMake modeling tool which allows to partition a workflow into several graph layers, instead of modeling all the workflow constraints on a single graph. The proposed approach also allows to verify the syntactic correctness of the process specifications by using a graph-reduction algorithm.

ADEPTflex [Reichert 1998] is also a graph-based modeling approach which allows ad hoc changes to process schema. The authors have proposed a complete and minimal set of change operations which that support users in modifying the structure of a workflow instance during its execution, while preserving structural correctness and consistency. The tool uses the correctness properties defined by the ADEPT to determine whether some specific change can be applied (or possibly rejected, if properties are violated) to the workflow instance. ActivityFlow [Liu 1997] provides a uniform workflow specification interface to describe different types of workflows and aims to increase the flexibility of workflow processes in accommodating changes. It also allows declarative and incremental flow specification and to reason about correctness and security of complex activities irrespective of their underlying implementation details. YAWL (Yet Another Workflow Language) is a Petri net based workflow language extended with additional features to facilitate the modeling of complex workflow and providing direct support for several patterns that are difficult to deal with using (colored) Petri net. It supports specification of the control flow and the data perspective of business processes and is supported by a software system that includes an execution engine, a graphical editor and worklist handler.

In terms of Web services composition and binding Web services to the activities in a graph (or workflow) based process model, proposed approaches can be classified as the static and dynamic workflow approaches as proposed in [Rao 2004a]. The static approaches require the process modeler to define an abstract process model using the graph-based modeling approaches as discussed above. Then the enactment and binding of Web services to the activities in the process model is handled either manually by defining the concrete Web service to be used to fulfill the activity or by automatic selection and discovery of Web service instances based on specified criteria by the process modeler. On the other hand, the dynamic workflow approaches [Casati 2000b, Schuster 2000] aim to both create the process model and selects atomic services automatically based on several constraints specified by the process modeler, including the dependency and control flow for the atomic services, the user's preferences and others. Further, in terms of implementing the business process and for its execution, the Web Services Business Process Execution Language (*WS-BPEL*) is the most commonly used approach.

3.1.2 AI Planning based composition model

A large number of approaches for the composition of Web services are based on AI (Artificial Intelligence) planning. The planning is a complex problem which has been investigated extensively by AI research, and it can be defined as a "kind of problem solving, where an agent uses its beliefs about available actions and their consequences, in order to identify a solution over an abstract set of possible plans" [Russell 1995]. In general, a planning problem includes the description of the possible actions which may be executed and their effects (a domain theory) in some formal language, the description of the initial state of the world and the description of the desired goal. The planner attempts to find a sequence of actions leading from an initial situation to the goal state. A planning problem can be formalized as a five-tuple (S, S_0, G, A, Γ), where S is the set of all possible states of the world, S_0 denotes the initial situation (or state), G denotes the goal state the planner attempts to reach, A is the set of actions the planner can perform in attempting to change from one state to another and eventually leading to goal state, and the relation Γ defines the precondition and effects for the execution of each action.

A planning problem includes domain formalization to provide a domain theory that represents the semantics of the actions and the proposed formalisms for specifying the domain theory can be broadly categorized as classical logics and extra-logical domain theories [Peer 2005]. The logical approaches include the situation calculus [Mccarthy 1963, Levesque 1998, Pirri 1999], event calculus [Kowalski 1986b] or the modal logics, as discussed by [Giacomo 1995, Giordano 1998, Castilho 1999]. However despite the advantages of these pure logic based approaches, such as precise semantics, the AI planning community has traditionally used formalisms based on STRIPS [Fikes 1971] notation, whose precise logical semantics have been a subject of debate. The ADL language [Pednault 1994] was then proposed to narrow the gap between the semantically ambiguous STRIPS and the declarative situation calculus and later Planning Domain Definition Language (PDDL) [Ghallab 1998] was developed to serve as a standard domain (and problem) specification language. Then, the basic planning paradigms include State-Space based Planning [McDermott 1996, Bonet 2001, Hoffmann 2001], Graph Based Planning [Blum 1997, Koehler 1997, Fox 2011], Partial Order Refinement Planning [Tate 1977, Penberthy 1992, Younes 2003], Planning as Satisfiability [Kautz 1992, Berardi 2003] and Planning as Logic Programming [Subrahmanian 1995, Shanahan 2000]. Other approaches have also been explored that aim to provide the planing agent with domain or task dependent control knowledge in order to achieve good performance in real world domains. Planning techniques that allow to incorporate and exploit domain or task-dependent control knowledge can be broadly categorizes into Hierarchical Task Network Planning , High-level Program Execution, Planning As Model Checking and Temporal Planning.

The interest in AI community for using AI planning for the Web services composition stems from the use of *OWL-S* (previously called DAML-S) for the semantically annotated Web services. "*OWL-S* is an ontology, within the OWL-based framework of the Semantic Web, for describing Semantic Web Services. It will enable users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints" [1]. The *OWL-S* ontology has three main parts: the service profile, the process model and the grounding. The process model includes the sets of inputs, outputs, pre-conditions and results of the service execution and this allows to translate the *OWL-S* based service description to a planning problem. Regarding the proposed AI planning based approaches for the Web services composition, we will first discuss the two different categories of approaches that aim to provide the planing agent with domain or task dependent control knowledge and then will briefly discuss the other AI-planning based approaches.

3.1.2.1 Golog and situation calculus

The classical approaches to planning require the planner to search a (possibly very large) space of possibilities to identify solution to the planning problem. An alternative approach is high-level program execution which transform the planning task to identify a sequence of actions which constitutes a legal execution of a given high level program. The Golog (alGOL in LOGic) [Levesque 1997] is such a high-level language. It is based on the situation calculus (a logical language for reasoning about action and change, [Mccarthy 1963, Levesque 1998, Pirri 1999]) providing a set of extra-logical constructs such as test actions, sequence, while loops and others. Golog based planning problems can be carried out by logic tools such as theorem provers. A variant of Golog capable of dealing with concurrency is ConGolog (Concurrent Golog) [Giacomo 2000].

In [McIlraith 2002] the authors advocate the use of a modified and extended version of ConGolog for the Web Service composition. The proposed approach is based on the notion of generic procedures and customizing user constraints and authors argue that the Web services composition can be viewed as customizations of reusable, Golog based high-level generic procedures instead of classical AI planning problem. As an example, the authors discuss the generic procedure to make travel plans and how it can be customized. Further, authors propose to archive generic procedures in sharable (DAML-S) ontologies so that multiple users can access and customize them. The authors have also proposed an implementation framework

[1]http://en.wikipedia.org/wiki/OWL-S

and suggest that their approach does not change the complexity of Web services composition, in contrast to traditional planning based approaches, and the proposed approach has the potential to reduce the search space, making it computationally advantageous. Further, in [Narayanan 2002b], a formal transformation from *OWL-S* to situation calculus and Golog is discussed and thus *OWL-S* processes can serve as specification of desired processes and the atomic and complex actions offered by Web services. The composition problem would then be to find an execution of a Golog program that satisfies the properties defined in the goal and are associated with the process.

3.1.2.2 Hierarchical Task Network (HTN) planning

Hierarchical Task Network (HTN) planning provides hierarchical abstraction to deal with the complex real world planning domains and was first introduced in the early Abstrips [Sacerdoti 1974] planner and in [Erol 1994] authors provide its formal semantics. HTN planning creates plans by task decomposition and assumes a set of operators (as similar to other planning systems) which are called primitive tasks. In addition, it also supports a set of methods, where each method specifies how to decompose some task into a set of subtasks. There are three types of goals in HTN planning as discussed in [Erol 1994]; goal tasks which are the properties of the goal state, the primitive tasks, and compound tasks that specify desired changes involving several goal tasks and primitive tasks.

An approach for using the HTN planning for the Web Services composition was proposed in [Sirin 2004] using the SHOP2 system [Nau 1999]. SHOP2 is a domain-independent planning system based on ordered task composition having the property that it plans for tasks in the same order in which they will be executed which can reduce the complexity of the planning process. The authors describe how SHOP2 can be used with *OWL-S* Web Service descriptions and provide a sound and complete algorithm to translate *OWL-S* service descriptions to a SHOP2 domain. The authors discuss that the *OWL-S* processes are pre-defined descriptions of actions to be carried out to get a certain task done as similar to HTN networks, and the concept of task decomposition in HTN planning is similar to the concept of composite process decomposition in *OWL-S* process ontology. The *OWL-S* composite processes serves as an input to a planner, which describes how to compose a sequence of single step actions and thus the goal of automated Web services composition is find a collection of atomic process instances which form an execution sequence for given composite process.

3.1.2.3 Other AI-planning based approaches

The strong interest from the AI community in Web services composition has lead to a number of AI-planning based approaches for the Web services composition. In this section we will briefly discuss some of the proposed approaches. In, SWORD

[Ponnekanti 2002] authors uses Entity-Relation (ER) model to specify the Web services. A Web service is represented in the form of a Horn rule that denotes the postconditions are achieve if the preconditions are true. They are specified in a world model that consists of entities and relationships among entities. To create a composite service, the requester only needs to specify the initial and final states for the composite service, then the plan generation can be achieved automatically using a rule-based expert system. In [McDermott 2002] authors introduce value of an action, which persists and which is not treated as a truth literal and which enables to distinguish the information transformation and the state change produced by the execution of the service. The input/output parameters for a service are considered to be reusable, thus the data values can be duplicated for the execution of multiple services. For instance, data values such as session IDs once established after some initial service invocations can then be re-used.

In [Sirin 2002] authors present a service selection approach based on OWL reasoner. The functionalities (parameters) are presented by OWL classes and OWL reasoner is applied to match the services, by checking if the output parameter of one service is the same OWL class or subclass of an input parameter of another service. The resulting matched services can also be ordered based on the distance between the two types in the ontology tree increases. In case of multiple-matches, it is further possible to filter the services based on the non-functional attributes. In [Medjahed 2003], authors propose a set of composability rules to determine whether two services are compatible and correspondingly composable. The composability rules include the syntactic rules (operation modes, binding protocols) and the semantic rules which may include checking the message, operation semantic, qualitative and soundness composability of the interacting Web services. The overall Web services composition approach first requires to specify the high-level description of the composition process using a language called Composite Service Specification Language(CSSL). Then, using the composability rules it is possible to generate composition plans that conform to specifications. In case of more than one plan, a particular plan is selected based on parameters such as rank, cost, and others. Then, the description of the composite service is automatically generated.

Further, various composition approaches rely on the theorem proving, as in [Waldinger 2000] authors elaborates an approach based on automated deduction and program synthesis. Available services and user requirements are described in a first-order language and then the Snark theorem prover is used and service composition descriptions are extracted from particular proofs. In [Sven 2002], authors apply a deductive approach for the synthesis of programs from specifications called Structural Synthesis of Program (SSP) for automated service composition. In [Rao 2003, Rao 2004b] authors introduces a method for automatic composition of semantic Web services using Linear Logic theorem proving.

3.1.3 Modeling non-functional requirements

In terms of modeling the non-functional requirements, a number of approaches have been proposed that extend and build upon the traditional approaches to model the non functional aspects. In this thesis, we will focus on the temporal and security aspects and as the Web services are autonomous, having local (temporal and security) constraints and as the composition process may have some global (temporal and security) constraints, the need to represent and compute an ordering satisfying the associated constraints is evident. The proposed approaches for incorporating temporal aspects include [Guermouche 2009a, Bordeaux 2004a, Ponge 2007, Benatallah 2005]. In [Kazhamiakin 2006], authors introduced a formalism called WSTTS to capture timed behavior of Web services and then using this formalism for model-checking *WS-BPEL* processes. In the planning domain, in [Tsamardinos 2003] authors proposed Conditional Temporal Problem (CTP) formalism that allows for the construction of conditional plans that satisfy complex temporal constraints. The approaches also include ISDL [Quartel 2004] which uses time attributes to represent properties. In order to verify the timed properties authors proposed converting the *WS-BPEL* process specification to timed automata and using *UPPAAL* model checker [Guermouche 2009b].

Further, there have been many approaches that aim to handle the security aspects in the Web services composition [Menzel 2009, Basin 2006, Garcia 2008, Souza 2009] however, as similar to the approaches for incorporating the temporal aspects, they focus on only 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 2009, Neubauer 2008, Rodríguez 2007]. Further, there have been approaches that aim to incorporate security requirements in the executable composition [Basin 2006, Garcia 2008, Chollet 2008] or their enforcement at execution time [Song 2006, Menzel 2009]. However, in [Souza 2009] authors proposed to use a formalism that allows for incorporating security aspects at different levels of abstraction and has important contributions in terms of identification of security requirements in the services composition.

3.1.4 Synthesis

Let us first briefly discuss the design principles for the process modeling before evaluating the proposed approaches discussed earlier. A process model is called procedural when it contains explicit and complete information about the flow of the process however, only implicitly keeps track of why these design choices have been made and if they are indeed part of the requirements or merely assumed for specifying the process flow [Goedertier 2008]. Although this adds a lot to the control over the composition process, however as there is tradeoff between the control and flexibility, this control comes at the expense of process flexibility and thus making the process rigid to adapt to continuously changing situations and possibly not even conforming to the process specification requirements. On the other hand, a process is termed as declarative when it models the minimal (and only specified) requirements that mark the boundary of the process and any suitable execution plan which meets the specified requirements is sought. For a declarative business process [Pesic 2006] concerns are made explicit, execution mechanism is goal driven instead of state driven and a declarative model allows for both design and run-time process modification. A detailed discussion about the declarative and procedural design approaches for the process specification can be found in [Goedertier 2008].

The graph-based composition modeling approaches are mostly procedural and although the graph-based approaches tend to be simpler and intuitive for the process modeler for the composition design, they over-constrain the process assuming the design choices that may not be present in the requirements but only added to specify the process flow. A typical example of such an approach is Business Process Modeling Notation (BPMN) and even of there is no dependency between the activities the process modeler is required to specify the process flow (possibly only using the sequence construct) that will result in over-constraining the process. A comparative analysis of proposed graph-based and rule-based approaches in terms of their expressibility, flexibility and adaptability, dynamism and complexity can be found in [Lu 2007] and the authors suggest that the rule-based approaches are able to express structure, data, execution, as well as temporal requirements and can model more Workflow patterns than the graph-based approaches. Further, rule-based approaches are indeed more flexible as they allow for the incomplete specification for task dependency and they provide better adaptability and dynamism support as the rule expressions can be revised at runtime to cater for the ad-hoc changes to process logic. However, one drawback the rule-based approaches is that they are harder to model as they have no visual appeal and modeling languages have a logical syntax and required some expertise when modeling.

The paradigm change from procedural to declarative process modeling was advocated in [Pesic 2006] by introducing the ConDec language for declarative process modeling. However, we believe that the traditional AI planning (and so as rule-based) approaches for composition process modeling are declarative and the advantages in terms of expressibility, flexibility and adaptability and dynamism we discussed above for the rule-based approaches can be generalized to declarative approaches as they are more expressive based and are based on formal logic and flexible as only constraints that mark the boundary of the process are specified.

Further, the traditional AI planning approaches such as Golog [McIlraith 2002] or HTN [Sirin 2004] based approaches are strongly related to *OWL-S* based process descriptions and the authors have proposed the formal semantics of *OWL-S* processes in situation-calculus considering how *OWL-S* processes can be transformed into situation calculus. This approach although very practical, limits the expressiveness of formalism as objective is to transform from *OWL-S* to situation calculus and not building about a formal design that can model *OWL-S* processes and also allows 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. Further, the need of verifying the composition process before execution and monitoring the process while in execution to cater and recover from unforeseen situation by finding alternatives is critical for the Web services composition. We believe that using a formal approach not only results in a highly expressive design approach but can also allow for the integration of different process life cycle stages using the same formalism. Further, it allows for formal verification of the composition process and allows for recovery actions such as re-planning or alternative path finding to cater for violations monitored during process execution.

3.2 Process verification

The traditional graph-based approaches for modeling the composition process, discussed in the previous section, are very intuitive and make it easier to model the processes however this ease is coupled with lack of strictness and the process specification includes arbitrariness allowing different interpretations for a single process. As a result a number of approaches have been approaches to define strict semantics to the processes in order to formally verify them and the composition process verification is highly active and widely studied research direction, Figure-3.2.

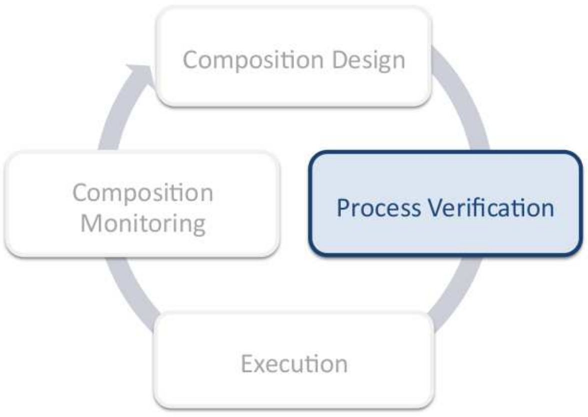


Figure 3.2: Web services composition process verification

The approaches for formal verification of the composition process can be broadly categorized into automata, Petri nets and process-algebra based approaches as discussed in [Morimoto 2008]. In this section, we will briefly discuss each category of the proposed approaches for the composition process verification.

3.2.1 Automata based approaches

Automata are base model of formal specifications for systems [Hopcroft 2001]. An automaton consists of a set of states and transition rules specify how to move from one state to another and can be regarded as a graph where nodes represent the states, the arcs between the nodes represent the transition from one state to another and labels on the arcs represent what actions cause the transition. Numerous approaches have been proposed to verify the Web services composition process using the automata based approaches and in general, automata based approaches require to first convert the composition process model (specified using *WS-BPEL*, *WS-CDL*) to automata. Then, proposed approaches require to convert the automata to XML formats to be used with model-checkers such as SPIN and *UPPAAL*.

In [Fu 2004], authors have extended the guarded automata model to allow the use of local XML variables and developed a tool which translates *WS-BPEL* web services to extended guarded automata model. These guarded automata can then be translated into Promela (Process or Protocol Meta Language) for the SPIN model checker [Holzmann 2004] to verify the properties (specified using Linear Time Temporal Logic) of composite web services specified in *WS-BPEL* and that communicate through asynchronous XML messages. In [Guermouche 2009b], authors propose a formal framework for analyzing the compatibility of a choreography in which the Web services support asynchronous timed communications. Timed properties are modeled as the standard clocks of standard timed automata and authors propose a set of required abstractions that allow to use the *UPPAAL* model checker to handle timed asynchronous services.

In [Díaz 2005], authors discuss how to translate Web services with time restrictions (described by *WS-BPEL*/WS-CDL) into a timed automata orchestration and then subsequently verify them by using the *UPPAAL* model checker[2]. The authors present an extensive Travel Reservation System case-study for discussing the translation and verification process using timed-automata. Further, in [Dong 2006] authors use the composition process specified using an approach called Orc, which has precise semantics and is proposed to support a structured way of orchestrating services. The authors define Timed Automata semantics for the Orc language and discuss how the Orc models are translated into timed-automata that can be verified using the *UPPAAL* model.

3.2.2 Petri net based approaches

Petri net is a framework to model concurrent systems and has an easily understandable graphical notation. As the traditional graph-based composition process modeling approaches can include arbitrariness and lack strictness allowing for different interpretation of same process models (or multiple models for the same pro-

[2]http://www.uppaal.com/

cess requirements), numerous modeling approaches have their origin in Petri net theory (or on different variants of Petri nets to provide extra expressibility and functionality) including High Level Petri nets [Ellis 1993], Low Level Petri nets [Wikarski 1996], and Colored Petri nets [Merz 1994], a detailed discussion can be found in [Janssens 2000]. In addition, different approaches investigate the use of Petri net for the composition process verification and in general the focus is how to translate business process diagrams into Petri net and once the translation has been done, a variety of Petri net based tools can be used for process verification however as discussed in [Morimoto 2008] various BPMN components such as gateways, event triggers, loop activities, are difficult to translate into Petri net.

The proposed Petri net base approaches include [Dijkman 2007] in which authors shows how to correspond BPMN constructs into labeled Petri net by providing a mapping from a subset of BPMN elements to Petri nets. The authors have also implemented the proposed framework however, the proposed mapping does not fully deal with some of BPMN constructs such as parallel multi-instance activities and OR-join gateways that highlight the limitations of Petri nets. In [Narayanan 2002a], the authors define the semantics of relation *WS-BPEL* and *OWL-S* in terms of situation calculus and they formalize business processes in Petri net, and have developed a tool to describe and automatically verify composition of business processes. In [Hamadi 2003], authors have proposed a Petri net-based algebra for composing Web services. The authors have provided a direct mapping from each composition operator to a Petri net construction and have claimed that any service composition expressed using the algebra constructs can be translated into a Petri net representation that also allows for the process verification.

In [Yi 2004], authors have proposed a Petri net-based design and verification framework that allows for both visualize and verify existing *WS-BPEL* processes and also provides support for creating new *WS-BPEL* processes. The framework enables the use of verification techniques at the design time and the generated *WS-BPEL* specification skeleton is thus verified process model. In [Zhang 2004], authors introduce a Petri net-based architectural description language called WS-Net, which is executable and incorporates the semantics of Colored Petri net and also supports the verification and monitoring of web services. WS-Net describes each Web services component in interface, interconnection and interoperation layers and the proposed approach requires manually transferring the WSDL specifications into the WS-Net specifications which the authors suggested is not trivial. Further, in [Hinz 2005] authors propose formal Petri net semantics for *WS-BPEL* processes which covers both standard behavior of *WS-BPEL* as well as the exceptional behavior such as faults, events and compensation, exception handling and compensations. In addition, the authors have implemented the proposed approach as a parser which can automatically convert *WS-BPEL* specification into the input language of the Petri net model checking tool LoLA to analyze the composition process.

3.2.3 Process Algebras based approaches

The process algebras are a family of approaches (including CSP, CCS, ACP, ambient calculus, fusion calculus, PEPA and LOTOS) for formally modeling the concurrent systems. They provide support for both specifying the high-level description of interactions, communications, and synchronizations between processes and algebraic laws that allow specified process descriptions to be analyzed. They also provide support for on bisimulation analysis (formal reasoning about equivalences between processes) which can be helpful to verify whether a service can substitute another service or the redundancy of a service [Bordeaux 2004b].

Proposed process-algebraic based approaches for Web services composition include [Salaün 2004], in which authors discuss the application of process algebras to, compose, and verify business processes. The authors have shown an example in which they use CCS to specify and compose business processes and the use of Concurrency Workbench[3] to verify the composition process. In [Ferrara 2004], the authors have proposed correspondence between *WS-BPEL* and LOTOS (including compensations and exception handling) and enables the verification of temporal properties with the CADP model checker.

3.2.4 Synthesis

The proposed approaches for the composition verification discussed in this section, in general, require mapping the process (mostly defined using procedural approaches such as *WS-BPEL*) to some formal logic (such as Petri nets, automata or process logic) and then using model checkers to verify the composition process. This transformation based approach has two major limitations. First, the proposed verification approaches are based on traditional procedural approaches and procedural approaches have less expressibility, flexibility and adaptability and dynamism as compared to the declarative ones. Further, the limited expressibility makes it difficult to verify the non-functional properties (such as temporal, security requirements or more importantly their combinations) associated with the composition process as for first, it is difficult to specify the non-functional properties using the traditional approaches such as *WS-BPEL* and a number of approaches have been proposed as an extension to *WS-BPEL* for specifying non-functional aspects and then it is not trivial (is possible) to add formal semantics to them for their verification.

Specifying the exact and complete sequence of activities to be performed for the composition process, as required by the traditional procedural approaches, however does make it possible to use proposed automata or Petri nets based approaches for design-time verification of composition process. However, with the declarative approaches the process may be only partially defined and thus this makes it difficult to use traditional approaches for the process verification as the transition system for a

[3]http://www.cs.sunysb.edu/ cwb/

declarative process can be very large. Further, the design-time verification should be coupled with execution-time monitoring and complexity of these approaches make them difficult to use for verifying the functional and non-functional constraints associated with the process while handle process change or recovery.

3.3 Process monitoring and recovery

The need to monitor the Web services composition process during execution stems from two major objectives. At one hand continuously monitoring the resource utilization, SLA's violation, or some domain specific Key Performance Indicators (KPI's) may be required to measure the performance or to fulfill some domain specific monitoring requirements. Then, as the Web services are autonomous and only expose their interfaces, composition process is based on design level service contracts and the actual execution of composition process may result in the violation of the design-level services contracts due to errors such as network or service failures, change in implementation or other unforeseen situation. This highlights the need to detect the errors and react accordingly to cater for them, Figure-3.3.

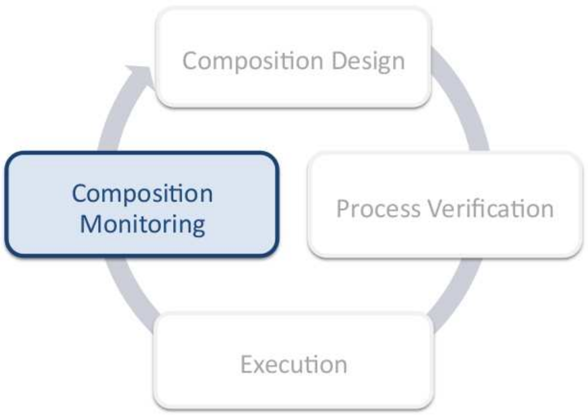


Figure 3.3: Process monitoring and recovery

The reaction may include to calculate the effect the violation has on the overall process execution and then to recover from it. Composition monitoring thus involves two related problems, how to effectively monitor the composition process in order to identify anomalies or for KPI's measurement and then once the violation is detected how to recover from the violation. In this section we will first briefly discuss the proposed approaches that focus on how to monitor the composition process and then the approaches that allow (possibly only partial) recovery. However, it is not always possibly to fully distinguish these related problems and approaches for monitoring may also handle process recovery and vice-versa.

3.3.1 Process monitoring

For the run-time monitoring of the composition process a number of approaches can be classified as assertion based approaches, as discussed in [Moser 2010]. In [Sun 2009], authors have proposed a monitoring approach based on Aspect-Oriented Programming (AOP) in order to verify the requirements specified using WS-Policy. In [Wu 2008], authors also present an AOP-based approach for identifying patterns (and to trigger corresponding actions) for the instances of *WS-BPEL* processes. In [Baresi 2005, Baresi 2011], authors have proposed a constraint language for the supervision of *WS-BPEL* processes called *WSCoL*. The approach builds upon *WS-BPEL* and the monitoring properties are specified as assertions on the *WS-BPEL* code. The authors then propose to use Aspect-Oriented Programming (AOP) to check the assertions wile the process is in execution. In [Baresi 2010], authors further extend their approach by separating data collection, aggregation and computation from the actual analysis. They have thus proposed a general monitoring language called *SECMOL*.

Then a number of approaches aim to monitor the temporal aspects in the Web services composition, in [Kallel 2009] authors propose an approach based on formal language called XTUS-Automata and in [Barbon 2006b], authors propose RTML (Runtime Monitoring Specification Language) and their approach translates monitoring information to Java code for monitoring.

In contrast to assertion based approaches, some event-based approaches have also been proposed [Moser 2010] and instead of using some language to define assertions, event-based approaches use CEP techniques to operate on event-streams, detect patterns and take corresponding actions, and combine events based on CEP operators such as temporal aspects. CEP techniques are very powerful because they operate on event streams (i.e., the monitoring data) and CEP operators are non intrusive. In [Suntinger 2008] authors propose a visualization approach to visualize complex event streams of historical information and allows to detect and analyze patterns. In [Beeri 2008] authors propose a query language allowing to visually design monitoring tasks.

3.3.2 Recovery

Traditional approaches for the Web services composition monitoring and recovery has roots in the exception handling approach provided by the Workflow management systems. For instance, in *WS-BPEL* different types of fault/event/message handlers and corresponding actions to be taken (based on the process state and ad hoc implementations of exception management) can be specified. Exception handling in general is an active and highly studied problem in WfMS has been and different types of exception handling mechanism are proposed [Russell 2006, Weske 2007, Vanhatalo 2007, Vanhatalo 2008]. However, most of them define handlers to be invoked under given conditions and provide support to handle given types of exceptions.

Regarding the proposed approaches for exception handling, in [Vanhatalo 2007, Vanhatalo 2008] authors propose an approach to not only identify the errors in work-flow models but also how the work flow models can be re-factored to support (partial) recovery from the identified errors. The SARN (Self-Adaptive Recovery Net) [Hamadi 2003] proposes a Petri net-based model and in [Cao 2005] authors propose a mobile-agent-based approach to handling exceptions. Pattern-based approaches have also been proposed as in [Russell 2006] authors present a pattern-based classification framework for exception handling while in [Casati 2000a, Casati 1999] propose exceptions handling based on predefined patterns derived from executions history.

As discussed earlier, the service based approaches in general build upon the exception handling approaches provided by the Workflow management systems (*Wfms*) to provide support for exceptions and handlers in service-based approaches. The motivation comes from the similarities in the Workflow based approaches and graph-based approaches fro composition process definition. In [Verma 2005] authors have presented an approach for elevating autonomic computing to the process level and have discuss the application of different self-* properties (self configuring, self healing, self optimizing and self aware) for Autonomic Web Processes (AWPs). In SCENE [Colombo 2006], the authors discuss that the problem to dynamically reconfigure composition processes needs to be addressed by having a flexible runtime platform and by having an expressive composition language that support constraints for the definition of rules for activating repair actions (e.g., alternative services, rebinding, and process termination). Authors partially address these issues by proposing a language for composition design that extends the standard *WS-BPEL* language with rules. In [Baresi 2007] authors enrich the *WS-BPEL* specification to propose self-healing processes by using supervision rules to annotate *WS-BPEL* processes and to react and recover (proposed recovery approaches include programmable,flexible, and extensible solution) in case of monitored violations.

The workflow based approaches discussed earlier propose an exception handling based solution to process monitoring and recovery. The AI planning based approaches on the other hand allow for plan revision, adaptation, and re-planning techniques to handle exceptions and process recovery.

3.3.3 Synthesis

Traditional approaches for the composition monitoring are proposed as an extension to some particular run-time and are tightly coupled and limited to it. In contrast the use of an event-based approach works on the message-level and thus is unobtrusive, independent of run-time and allows for integration of other systems and processes, as discussed in [Moser 2010]. Then, the traditional monitoring approaches [Barbon 2006a, Baresi 2009, Mahbub 2004] build upon composition frameworks that are highly procedural, such as *WS-BPEL*, and this in-turn poses two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not part of the composition framework and functional and non-functional properties are not expressed in terms of events and their effects. Secondly the use of procedural approach for process specification does not bridge the gap between organization and situation in a way that it is very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time, and it does not allow for a reasoning approach allowing for effects calculation and recovery actions such as re-planning or alternate path finding as we discussed in [Zahoor 2010a].

In [Moser 2010] authors proposed an event based monitoring approach that works on the message-level and thus is unobtrusive, independent of run-time and that highlights the need and motivation for using an event-based monitoring framework. However the approach aims to extract and define events from procedural process specification, while our approach builds on an event-based framework and events are first class objects in both composition design and monitoring framework. This allows to reason about events during execution and allows for effects calculation and different types of recovery actions. In [Kowalski 1986a] authors attempt to add monitoring directives to a declarative approach but still the approach lacks expressiveness and does not allow for recovery actions. Our work can be compared to PAWS framework [Ardagna 2007], in which authors propose to add annotations to the *WS-BPEL* process to handle services replacement in case of run-time failure. However, as the approach is based on *WS-BPEL* and is procedural, it allows for limited recovery options (such as service replacement and not re-planning or alternative path finding) and effects calculation once a violation is detected. In [Friedrich 2010] authors proposed a model based approach for repair by exploiting information about the causes of process and deriving repair strategies based on process structure, however the approach builds upon *WS-BPEL* and *PAWS* and thus does not allow to reason about monitoring properties and allowing for effects calculation and different recovery schemes.

3.4 Summary

The proposed event-based approach for Web services composition presented in this thesis both aims to integrate different stages of process life-cycle, Figure-1.2, and advocates the use of declarative event-based approach at different stages. In this chapter, we have thus discussed categories of proposed approaches in the literature for each process life cycle stage and provided a detailed synthesis for the limitations they pose.

For the composition design, different approaches have been proposed having their strengths and weaknesses in terms of ease of usage and expressibility of the modeling language. Further, as it is important to have a process model properly defined, verified, and refined before being implemented, different approaches offer varying level of complexity for model-checking the composition process. The proposed approaches for the composition process modeling can be broadly categorized into Workflow based composition and AI planning based approaches, as discussed in [Rao 2004a] or similarly as either graph-based or rule-based approaches as discussed in [Lu 2007].

The proposed graph-based composition modeling approaches are mostly procedural and although the graph-based approaches tend to be simpler and intuitive for the process modeler for the composition design, they over-constrain the process assuming the design choices that may not be present in the requirements but only added to specify the process flow. A comparative analysis of proposed graph-based and rule-based approaches in terms of their expressibility, flexibility and adaptability, dynamism and complexity can be found in [Lu 2007] and the authors suggest that the rule-based approaches (and so as declarative approaches) are able to express structure, data, execution, as well as temporal requirements and can model more Workflow patterns than the graph-based approaches. Further, rule-based approaches are indeed more flexible as they allow for the incomplete specification for task dependency and they provide better adaptability and dynamism support as the rule expressions can be revised at runtime to cater for the ad-hoc changes to process logic. However, one drawback the rule-based approaches is that they are harder to model as they have no visual appeal and modeling languages have a logical syntax and required some expertise when modeling. The paradigm change from procedural to declarative process modeling was advocated in [Pesic 2006] by introducing the ConDec language however we believe that the traditional AI planning (and so as rule-based) approaches for composition process modeling are declarative.

Further, the traditional AI planning approaches such as Golog [McIlraith 2002] or HTN [Sirin 2004] based approaches are strongly related to *OWL-S* based process descriptions and the authors have proposed the formal semantics of *OWL-S* processes in situation-calculus considering how *OWL-S* processes can be transformed into situation calculus. This approach although very practical, limits the expressiveness of formalism as objective is to transform from *OWL-S* to situation calculus and not building about a formal design that can model *OWL-S* processes and also allows 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.

For the composition process verification, some seminal work on categorizing the verification properties and defining correctness of the composition process can be found in [Röglinger 2009].The authors have categorized the verification properties in application dependent and application independent categories and have also classified them as structural or behavioral correctness categories. The approaches for formal verification of the composition process can be broadly categorized into automata, Petri nets and process-algebra based approaches as discussed in [Morimoto 2008].

The proposed approaches for the composition verification, in general, require mapping the process (mostly defined using procedural approaches such as *WS-BPEL*) to some formal logic (such as Petri nets, automata or process logic) and then using model checkers to verify the composition process. This transformation based approach has two major limitations, first the proposed verification approaches are based on traditional procedural approaches and procedural approaches have less expressibility, flexibility and adaptability and dynamism as compared to the declarative ones. Further, the limited expressibility makes it difficult to verify the non-functional properties (such as temporal, security requirements or more importantly their combinations) associated with the composition process as for first, it is difficult to specify the non-functional properties using the traditional approaches such as *WS-BPEL* and a number of approaches have been proposed as an extension to *WS-BPEL* for specifying non-functional aspects and then it is not trivial (is possible) to add formal semantics to them for their verification.

Further, specifying the exact and complete sequence of activities to be performed for the composition process, as required by the traditional procedural approaches, however does make it possible to use proposed automata or Petri nets based approaches for design-time verification of composition process, however with the declarative approaches the process may be only partially defined and thus this makes it difficult to use traditional approaches for the process verification as the transition system for a declarative process can be very large. Further, the design-time verification should be coupled with execution-time monitoring and complexity of these approaches make them difficult to use for verifying the functional and non-functional constraints associated with the process while handling process change or recovery.

Then, in order to monitor the composition process while in execution we have discussed proposed approaches for two related problems; first how to effectively monitor the composition process in order to identify anomalies or for KPI's measurement and then once the violation is detected how to recover from the violation. Traditional approaches for the composition monitoring are proposed as an extension to some particular run-time and are tightly coupled and limited to it. In contrast the use of an event-based approach works on the message-level and thus is unobtrusive, independent of run-time and allows for integration of other systems and processes, as discussed in [Moser 2010].

In addition, the traditional monitoring approaches [Barbon 2006a, Baresi 2009, Mahbub 2004] build upon composition frameworks that are highly procedural, such as *WS-BPEL*, and this results in having two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not part of the composition framework and functional and non-functional properties are not expressed in terms of events and their effects. Secondly the use of procedural approach for process specification does not bridge the gap between organization and situation in a way that it is very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time, and it does not allow for a reasoning approach allowing for effects calculation and recovery actions such as re-planning or alternate path finding as we discussed in [Zahoor 2010a]. For process recover, our work can be compared to PAWS framework [Ardagna 2007], in which authors propose to add annotations to the *WS-BPEL* process to handle services replacement in case of run-time failure. However, as the approach is based on *WS-BPEL* and is procedural, it allows for limited recovery options (such as service replacement and not re-planning or alternative path finding) and effects calculation once a violation is detected. In [Friedrich 2010] authors proposed a model based approach for repair by exploiting information about the causes of process and deriving repair strategies based on process structure, however the approach builds upon *WS-BPEL* and PAWS and thus does not allow to reason about monitoring properties and allowing for effects calculation and different recovery schemes.

Part III

COMPOSITION DESIGN

Chapter 4

Composition design - components

Contents

The composition process modeling is the first and most important stage of the composition process life cycle, Figure-3.1. The objective of composition process modeling is to provide high-level specification independent from its implementation that should be easily understandable by process modeler who create 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 event-calculus.

In this chapter, we will discuss event-calculus models for different components in reference to the proposed framework. The various components that constitute the composition design can be broadly divided into **activity** and **service** categories. Activity is a general terms for any work being performed while 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.

4.1 Activities

Activity is a general term for representing any work being performed[1]. In this section we will discuss event-calculus models for modeling different aspects related to activities. We will also provide an example for using the proposed event-calculus based activity model, in order to introduce the reader to the proposed reasoning approach.

4.1.1 Activities with states

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 event-calculus we can define events that represent the actions required to *start* and *finish* the activities and the activity state can be represented by defining event-calculus fluents. The event-calculus formalism below models activities:

Activities (with states) *- activitywithstate.e*
Usage: *- import activitywithstate.e and define instances of sort activitywithstate to define activities*

```
sort activityS

fluent Started_ activityS(activityS)
fluent Finished_ activityS(activityS)
event Start_ activityS(activityS)
event End_ activityS(activityS)

Initiates(Start_ activityS(activityS), Started_ activityS(activityS),time).
Initiates(End_ activityS(activityS), Finished_ activityS(activityS),time).

Terminates(End_ activityS(activityS), Started_ activityS(activityS),time).

Happens(End_ activityS(activityS), time) → HoldsAt(Started_ activityS(activityS), time).
HoldsAt(Started_ activityS(activityS), time) → !Happens(Start_ activityS(activityS), time).
HoldsAt(Finished_ activityS(activityS), time) → !Happens(End_ activityS(activityS), time).

HoldsAt(Finished_ activityS(activityS), time) → !Happens(Start_ activityS(activityS), time).

!HoldsAt(Started_ activityS(activityS), 0).
!HoldsAt(Finished_ activityS(activityS), 0).
```

In the model above, we first defined an event-calculus sort named *activityS* and instances of the sort will represent the actual activities[2]. 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 event-calculus 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. Further, as a result of *End*

[1] Activities are also termed as tasks but we will use the term activity.

[2] As DECReasoner language does not allow upper-case alphabets in sort names, the actual event-calculus model has a sort named *activitywithstate* abbreviated as *activityS*.

event, the activity state changes to *Finished* (represented by the second *Initiates* axiom) and we have also defined 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).

Further, we have defined some axioms to control the invocation of specified events such as the *End_ activityS* event should only *Happen* once the activity has already been started, and the fluent *Started_ activityS 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.*

4.1.2 Example

Before going further, let us briefly discuss how such an event-calculus based composition model can be used for reasoning by defining two instances (*ActivityA* and *ActivityB*) of sort *activityS.* In the model below, we first import the event-calculus core files (*root.e* and *ec.e*) and then we import the *activitywithstate.e* file which contains the event-calculus model for activities modeling shown earlier. Then, we define instances of sort *activitywithstate* that represent the activities and also the goal for the process that is to have the activities *Finished* (and thus requiring the fluent *Finished_ activitywithstate* 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).*

Activities definition using activitywithstate.e

```
;including helper files
load foundations/Root.e
load foundations/EC.e
load includes/activitywithstate.e

;creating instances representing activities
activitywithstate ActivityA, ActivityB

;initial conditions for the fluents
!HoldsAt(Finished_ activitywithstate(activitywithstate), 0).
;composition goal
HoldsAt(Finished_ activitywithstate(activitywithstate), 2).

range time 0 2
range offset 1 1
option showpred off
```

Invoking the event-calculus reasoner for the above instantiated model gives us the solution shown below. The solution returned by the reasoner shows that if the *Start_ activitywithstate* events happen (representing that the activities are thus being started) at time-point *0*, the fluents *Started_ activitywithstate* 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*). Further, once the activities have been started the *End_ activitywithstate* events happen at time-point *1* to have the fluents *Finished_ activitywithstate* hold at time-point *2*, that was the specified process goal. Notice that the *End_ activitywithstate* events also make the fluents *Started_ activitywithstate* does not hold as indicated by the $-$ sign shown next to them at time-point *2*, representing the activity state is no longer *Started*.

Activities definition using activitywithstate.e

```
Discrete Event Calculus Reasoner 1.0
loading activityS_ instances.e
loading foundations/Root.e
loading foundations/EC.e
loading includes/activitywithstate.e
32 variables and 78 clauses
relsat solver
1 model
—
model 1:
0
Happens(Start_ activitywithstate(ActivityA), 0).
Happens(Start_ activitywithstate(ActivityB), 0).
1
+Started_ activitywithstate(ActivityA).
+Started_ activitywithstate(ActivityB).
Happens(End_ activitywithstate(ActivityA), 1).
Happens(End_ activitywithstate(ActivityB), 1).
2
-Started_ activitywithstate(ActivityA).
-Started_ activitywithstate(ActivityB).
+Finished_ activitywithstate(ActivityA).
+Finished_ activitywithstate(ActivityB).

encoding 0.0s
solution 0.0s
total 0.1s
```

The activity model shown in this section can be further enriched to specify other related aspects, such as the time taken by an activity or dependency between activities, as we will discuss later in Section-5.1.

4.1.3 Activities without intermediate states

The activities may not always need to be represented by the states and a simpler version of the previous activity model is shown below (*activitywithoutstate* abbreviated as *activityWS*):

Activities (without states) *- activitywithoutstate.e*
Usage: *- import activitywithoutstate.e and define instances of sort activitywithoutstate to define activities*

sort activityWS
event Start_ activityWS(activityWS)
fluent Finished_ activityWS(activityWS)
Initiates(Start_ activityWS(activityWS), Finished_ activityWS(activityWS),time).
HoldsAt(Finished_ activityWS(activityWS), time) → !Happens(Start_ activityWS(activityWS), time).
!HoldsAt(Finished_ activityWS(activityWS), 0).

4.1.4 Activities that can be restarted

The activity models shown so far in this section does not cater for the cardinality constraints and thus does not allow to re-start an activity once it has finished. However, the activities may require to be restarted (as in case of activities within loop body) and in order to model the activities that can be restarted we enrich the activity model as below (*activitywithstaterestart* abbreviated as *activitySR*) :

Activities (with states that can be restarted) *- activitywithstaterestart.e*
Usage: *- import activitywithstaterestart.e and define instances of sort activitywithstaterestart to define activities*

sort activitySR

fluent Started_ activitySR(activitySR)
fluent Finished_ activitySR(activitySR)
event Start_ activitySR(activitySR)
event End_ activitySR(activitySR)

event ResetactivitySRStatus(activitySR)

Initiates(Start_ activitySR(activitySR), Started_ activitySR(activitySR),time).
Initiates(End_ activitySR(activitySR), Finished_ activitySR(activitySR),time).
Terminates(End_ activitySR(activitySR), Started_ activitySR(activitySR),time).

Terminates(ResetactivitySRStatus(activitySR), Started_ activitySR(activitySR),time).
Terminates(ResetactivitySRStatus(activitySR), Finished_ activitySR(activitySR),time).

Happens(End_ activitySR(activitySR), time) → HoldsAt(Started_ activitySR(activitySR), time).
Happens(ResetactivitySRStatus(activitySR), time) → HoldsAt(Finished_ activitySR(activitySR), time) .
HoldsAt(Started_ activitySR(activitySR), time) → !Happens(Start_ activitySR(activitySR), time).
HoldsAt(Finished_ activitySR(activitySR), time) → !Happens(End_ activitySR(activitySR), time).
HoldsAt(Finished_ activitySR(activitySR), time) → !Happens(Start_ activitySR(activitySR), time).

!HoldsAt(Started_ activitySR(activitySR), 0).
!HoldsAt(Finished_ activitySR(activitySR), 0).

In the model above, we have enriched the activity with states model to add an event called *ResetactivitySRStatus* and have defined *Terminates* axioms for the fluents, *Started_ activitySR* and *Finished_ activitySR*. As a result, once the *ResetactivitySRStatus* event happens it resets the status of activity and thus allows to re-start an activity. Further, we can also define when the *ResetactivitySRStatus* event happens as we will discuss later while defining models for control flow constructs such as loops.

4.2 Web services

The proposed composition design also allows to model the Web services and in this section we will discuss event-calculus models for Web services supporting different invocation modes (synchronous or asynchronous). As the proposed framework aims to reason about the composition process, we will only model the core aspects related to the Web services that are needed to be reasoned about and leaving out the details only needed for services execution. However, proposed models can be extended to handle other aspects, if needed.

4.2.1 Synchronous Web services invocation

In order to model the synchronous Web services invocation, we can define an event-calculus sort called *synchservice* and its instances represent the Web services used in the composition process. We can then define events to invoke the Web services and fluents that represent if the response has been received from the Web service. The event-calculus model below allows synchronous Web services invocation:

Web services (synchronous invocation without delay) - *synchservice.e*
Usage: *- import synchservice.e and define instances of sort synchservice to represent Web services*

```
sort synchservice

fluent ResponseReceived_synchservice(synchservice)
event Invoke_synchservice(synchservice)

Initiates(Invoke_synchservice(synchservice), ResponseReceived_synchservice(synchservice),time).
HoldsAt(ResponseReceived_synchservice(synchservice), time) → !Happens(Invoke_synchservice
(synchservice),time).

!HoldsAt(ResponseReceived_synchservice(synchservice),0).
```

In the event-calculus model above, we define an event to specify the service invocation *Invoke_synchservice*, a fluent *ResponseReceived_synchservice*, which specifies if we have received the response message from the Web service and an *Initiates* axiom that states if the action *Invoke_synchservice*, happens at some time then the fluent *ResponseReceived_synchservice* continues to hold after that time. We also define that once the service has been invoked and response received, it should not be re-invoked again and the initial condition for the fluent that it does not hold at time-point 0.

The synchronous Web service model discussed above is simplistic model that does not allow for specifying the delay a Web service takes to produce response. In order to model the response time delay for a Web service (see *synchservicewithdelay.e* file), we can break down the Web service invocation event *Invoke_synchservice*, into two separate events (*Invoke* and *EndInvocation*) and add corresponding fluents. Then we can define the delay between invocation start and end events to model response time delay.

4.2.2 Pull-based Asynchronous invocation

The invocation mode for the Web services can be pull-based asynchronous and the composition process can request and later "pull" the response message from the Web service after some specified delay. In order to model the pull-based asynchronous invocation, we can update the event-calculus model for synchronous Web services invocation by adding events and fluents for the sending request and then pulling the response, as shown in the model below (*asynchpullservice* abbreviated as *APull*).

Web services (Pull-based asynchronous invocation) - *asynchpullservice.e*
Usage: - *import asynchpullservice.e and define instances of sort asynchpullservice to represent Web services*

```
sort APull
fluent ResponseReceived_APull(APull)
fluent ResponseRequested_APull(APull)

event Invoke_APull(APull)
event EndInvocation_APull(APull)

Initiates(Invoke_APull(APull), ResponseRequested_APull(APull),time).
Initiates(EndInvocation_APull(APull), ResponseReceived_APull(APull),time).
Happens(EndInvocation_APull(APull),   time)→   HoldsAt(ResponseRequested_APull   (APull),
time).

HoldsAt(ResponseRequested_APull(APull), time) → !Happens(Invoke_APull (APull),time).
HoldsAt(ResponseReceived_APull(APull), time) → !Happens(EndInvocation_APull (APull),time).

!HoldsAt(ResponseReceived_APull(APull),0).
!HoldsAt(ResponseRequested_APull(APull),0).
```

4.2.3 Push-based Asynchronous invocation

The invocation mode for the Web services can also be push-based asynchronous and the composition process can request and the response is later "pushed" by the service provider to the composition process. In order to model the push-based asynchronous invocation, we introduce the 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. the event-calculus model below handles the push-based asynchronous invocation for the Web services ((*asynchpushservice* abbreviated as *APush*)). The process first sends the request to the Web service (as similar to the previous models, using the event *Invoke_APush*) and then the response is pushed to the process queue, *PushResponse_APush*, between some specified time intervals. Once the data is available in the queues, *HoldsAt(ResponsePushed_APush)*, the response can then be retrieved from the process queue using the event *EndInvocation_APush*.

Web services (Push-based asynchronous invocation) *- asynchpushservice.e*
Usage: *- import asynchpushservice.e and define instances of sort asynchpushservice to represent Web services*

```
sort APush

fluent ResponseReceived_APush(APush)
fluent ResponseRequested_APush(APush)
fluent ResponsePushed_APush(APush)

event Invoke_APush(APush)
event PushResponse_APush(APush)
event EndInvocation_APush(APush)

Initiates(Invoke_APush(APush),ResponseRequested_APush(APush),time).
Initiates(PushResponse_APush(APush),ResponsePushed_APush(APush),time).
Initiates(EndInvocation_APush(APush),ResponseReceived_APush(APush),time).

Happens(PushResponse_APush(APush),time)→ HoldsAt(ResponseRequested_APush(APush),time).
Happens(EndInvocation_APush(APush),time)→ HoldsAt(ResponsePushed_APush(APush),time).

HoldsAt(ResponseRequested_APush(APush), time) → !Happens(Invoke_APush(APush),time).
HoldsAt(ResponsePushed_APush(APush), time) → !Happens(PushResponse_APush(APush),time).
HoldsAt(ResponseReceived_APush(APush), time)→ !Happens(EndInvocation_APush(APush),time).

!HoldsAt(ResponseReceived_APush(APush),0).
!HoldsAt(ResponsePushed_APush(APush),0).
!HoldsAt(ResponseRequested_APush(APush),0).
```

4.2.4 Services re-invocation

The Web services models discussed in this section does not allow to re-invoke a service once it has been invoked and the response has been received. In order to handle the services re-invocation (which would be needed for defining the iteration control construct, Section-5.4) we can add *terminates* axioms, that reset the fluents representing that the service has been invoked and the response message has been received. The event-calculus model below is for the synchronous Web services invocation that can be re-invoked (substituting *synchservicewithreinvoke* for *serviceSR*):

Web services (synchronous invocation that can be re-invoked) - *synchservicewithreinvoke.e*
Usage: - *import synchservicewithreinvoke.e and define instances of sort synchservicewithreinvoke to represent Web services*

```
sort serviceSR

fluent ResponseReceived_serviceSR(serviceSR)
event Invoke_serviceSR(serviceSR)
event ResetStatus_serviceSR(serviceSR)

Initiates(Invoke_serviceSR(serviceSR), ResponseReceived_serviceSR(serviceSR),time).
HoldsAt(ResponseReceived_serviceSR(serviceSR),time) → !Happens(Invoke_serviceSR (serviceSR),time).
Terminates(ResetStatus_serviceSR(serviceSR), ResponseReceived_serviceSR(serviceSR),time).
Happens(ResetStatus_serviceSR(serviceSR), time) → HoldsAt(ResponseReceived_ serviceSR(serviceSR), time) .

!HoldsAt(ResponseReceived_serviceSR(serviceSR),0).
```

4.3 Nodes

In addition to the Web service instances already known, the user can also add abstract Web service types, called nodes, that need to be discovered and instantiated to some concrete Web service instances based on some specified constraints. Each node has a unique type, that specifies the type of Web services that can be discovered and bound to the node. The constraints associated with the nodes can include functional and non-functional properties needed for services discovery.

In order to model nodes using event-calculus, we add the sorts *node* and *constraints* (which specify the constraints added to a node). 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.

Further, we 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 axioms to only handle the invocation for the concrete Web services (and not to invoke the services is repository unless they are bound to some Nodes and are made concrete).

Web service nodes - nodes.e
Usage: -

```
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.

Web service nodes - nodes.e
Usage: -

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

HasConst(node, constraint) & !SatisfiesConst(service, constraint) → !Happens(Bind(node, ser-
vice),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) & Hold-
sAt(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 Section-7.1 for a SQWRL based approach that searches through the OWL-S based repository using SQWRL queries) may be a better option. In this approach, the nodes are discovered and the candidate services for a node are added to the event-calculus with the axioms that exactly one service is executed (see [Zahoor 2009b] for an example).

4.4 BPMN and event-calculus

In this section we will briefly discuss how the proposed event-calculus based approach relates to the BPMN by both relating the proposed models to BPMN core elements and also considering a mapping from core BPMN elements to the event-calculus. The objective is twofold; at one hand this allows to provide formal-logic semantics to the BPMN elements while on the other hand, the resulting design is highly expressive, flexible and allows for reasoning about the BPMN elements to identify conflicts and to compute an ordering based on (possibly) partially defined process.

An overview of the mapping BPMN core elements in event-calculus is shown in the table below. In general the different BPMN elements can be regarded as event-calculus *sorts* (which are types) and a particular BPMN entity of some element type can be regarded as the instance of event-calculus sort. Further, we can also have sub-sorts in the event-calculus that allows to model hierarchy among BPMN elements and for instance, providing roles-hierarchy in a role based access control system. In order to model properties of and relationships between different elements we can correspondingly create event-calculus predicates that can than be used to reason about them.

Element	*Event-calculus model*
Event	*Instances of basic EC sort event, their trigger can be defined as axioms and their impact can be defined by effect on fluents.*
Activity	*Can be mapped directly to Activities model we presented in Section-4.1.*
Gateway	*BPMN gateway constructs can be mapped to proposed Split/Join constructs, Section-5.2.*
Sequence/Message flow	*Sequence flow corresponds to the proposed dependency construct (Section-5.1) with some difference as discuss below. Message flow can be mapped to the proposed message flow construct discussed in Section-5.6*
Association	*We can have a predicate* HasAssociation(parameters) *to define associations for different elements.*
Pool/Lane/Group	*We can have sorts called pool, lane, group and can use EC predicates (similar to the association) to specify which activity belong to which pool/lane/group.*
Data object/ Message/Text	*Can be mapped to the request/response data elements modeling presented in Section-5.5.*

Events are the core concept in the proposed modeling approach and in terms of event-calculus terminology they can be regarded as the instances of basic EC sort *event*, each event has a trigger and impact. Their trigger can be defined as axioms that define the necessary conditions for the events to happen and their impact can be defined by the effect on fluents and/or the triggering of other events. Events can have parameters and can have multiple (combination of) impacts and triggers. In relation to the events defined in the BPMN, we do not distinguish between start, intermediate and end events. All the events can have impact(s) and trigger(s). In order to mark the entry and exit point of the process we can have events named Start and End, however in contrast to BPMN notation the End event can also have impact(s), for instance changing the process status fluent, *ProcessTerminated()*. Some examples of events used in the proposed model include the events to invoke Web services (with different synchronization modes), events to handle data flow, temporal and security aspects and others.

Activities as proposed in BPMN notation, can be mapped directly to the activities models we presented in Section-4.1. Regarding the ***control/data flow*** BPMN constructs modeling using event-calculus, the sequence (message) flow BPMN element can be mapped as specifying the (data) ***dependency*** (Section-5.1) between different components with one major difference; in BPMN notation, a sequence flow is used to show the order in which the activities will be performed in a process which may or may not be based on dependency. As a result, the process gets over-constrained and becomes highly procedural. In contrast the proposed approach allows to specify the dependency between components if and only if, there exists the dependency. One important advantage of the proposed approach is that it allows to define events-based dependency and thus dependency can be specified between any two events. This in-turn, makes it possible to specify dependency not only on the successful completion of a component but also on the partial state of a component. Examples include to define the dependency on a activity being started, in execution or on a service being invoked (and not yet completed) or on the data, such as data has been received, expired or the reception of some particular data values.

For the BPMN ***gateways***, the proposed XOR Split (Section-5.2) is an Exclusive gateway, the OR split is Inclusive gateway while the AND-Split is the Parallel gateway. The proposed event based split is different to the BPMN event-based gateway in a way that the split decision is based on the occurrence or absence of the events, while the BPMN event-based gateway can be converted to a data-based split gateway. The ***association*** BPMN construct is used to associate information and artifacts with the data elements. In event-calculus this can be modeled by defining a predicate called *HasAssociation(artifact, activity)* to define associations for different artifacts, that can be instances of a defined sort named artifact. Then for the Pool/Lane/Group we can have sorts called pool, lane, group and can use EC predicates (similar to the association) to specify which activity belong to which pool/lane/group. For the Data object/Message/Text we can have a sort object/message/text and the predicate HasDataObject(activity, object).

4.5 Example

We will now review the motivating example (Section-2.2) and discuss event-calculus modeling for different aspects related to the example. In this section we will only identify different components regarding the motivating example and leave the detailed discussion about modeling the control/data flow for Chapter-5 and about modeling the non-functional requirements for Chapter-6. The event calculus model below lists the components regarding the motivating example:

Motivating example - Defining components

```
load foundations/Root.e
load foundations/EC.e
load includes/synchservice.e
load includes/activitywithstate.e
load includes/activitywithoutstate.e
synchservice FireBrigadeWS, CallStaffWS, PoliceWS, AmbulanceWS, ExternalOrgWS, MeteorologyDeptWS
activitywithstate FireContainment, ExamineSite, PlanRecovery, RecoverPriorityItems
activitywithoutstate Start, End1, End2
```

For modeling different components for the motivating example using proposed event-calculus based approach we first define the generic models to be included (such as includes/synchservice.e and others) and instances of different sorts for defining components constituting the composition process, as shown in the model above.

4.6 Summary

Different components that constitute the composition process can be broadly categorized into **activities** and **services** categories. In this chapter, we have first presented how activities (with intermediate states) can be modeled using event-calculus and then used that activity model to present an example highlighting how the model can be used for reasoning purposes. We have then presented different sub-models for activities such as the one for activities that do not require intermediate states and for the ones that need to be restarted (probably within a loop-body). The table below provides an overview regarding the structure of this chapter for modeling activities.

Activities	***Section for event-calculus models***
Activities with states	*Event-calculus model is discussed in Section-4.1.1, in order to use the model include activitywithstate.e to the event-calculus file.*
Instantiated model	*A basic example showing how to use the activitywithstate.e for reasoning using DECReasoner is shown in Section-4.1.2.*
Without states	*A simplified model, where activity states are not needed is discussed in Section-4.1.3 and corresponding include file is activitywithoutstate.e.*
Activities with restart	*Activities that need to be restarted are modeled in Section-4.1.4 and corresponding include files are activity(with/without)staterestart.e*

For the Web services, we have presented event-calculus based models for Web services supporting different synchronization modes (synchronous, push/pull based asynchronous). We have also detailed how event-calculus based approach can be also be used to modeling Nodes highlighting the limitations it poses and leaving the discussion to Section-7.1 for an alternative SQWRL-based approach for Nodes. The table below provides an overview regarding the structure of this chapter for modeling different Web service types.

Web services	***Section for event-calculus models***
Synchronous	*Web services model with synchronous invocation mode is discussed in Section-4.2.1. Corresponding include files are synchservice(withdelay).e.*
Asynchronous (pull)	*Pull-based asynchronous Web services invocation model is presented in Section-4.2.2, corresponding include file is asynchpullservice.e.*
Asynchronous (push)	*Push-based asynchronous invocation model is discussed in Section-4.2.3, corresponding include file is asynchpushservice.e.*
Services(re-invoke)	*EC model for Web services that need to reinvoked (for instance within loop body) is discussed in Section-4.2.4, corresponding include files are (asynchpull/asynchpush/synch)servicewithreinvoke.e*
Nodes	*An brief discussion about the nodes is presented in Section-4.3.*

The event-calculus models presented in this section are organized into generic self-contained models added to independent event-calculus files that can be included into process specification. For instance, in order to add some synchronous Web services to the process specification we can simply include the file called, synchservice.e, and define instances of event-calculus sort called *synchservice*. This generic approach has allowed to implement a Java-based application, called *ECWS*, that can automatically generate the event-calculus models for the process specification, we will discuss the *ECWS* application in Section-9.2. We have also identified different components for the motivating example, presented in Section-2.2 and discussed event-calculus models for representing the components.

Chapter 5

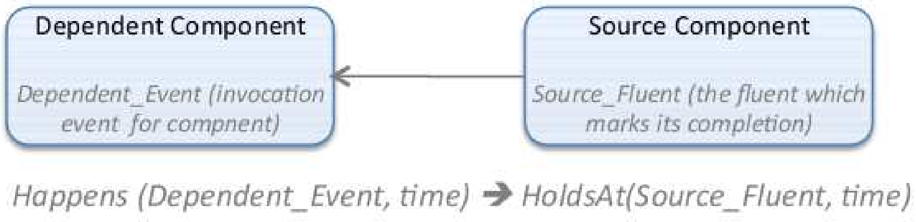
Control/Data flow specification

Contents

In this chapter we will discuss different control and data flow constructs modeling using event-calculus, that specify the dependencies and control and data flow between different components[1] added to the composition design.

5.1 Dependency

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, Figure-5.1.



Dependent Component
Dependent_Event (invocation event for compnent)
Source Component
Source_Fluent (the fluent which marks its completion)
Happens (Dependent_Event, time) ➔ HoldsAt(Source_Fluent, time)

Figure 5.1: The dependency construct

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

[1]We will use the generic term component that can either be a service or an activity.

invocation/start event of the dependent component should not happen unless the fluent representing the completion of the source component holds. We can thus define the following generic pattern:

Pattern for specifying dependency construct
Usage: see example below

;Dependency specification without explicitly defining the delay
Happens(DEPENDENT_EVENT, time) → HoldsAt(SOURCE_FLUENT, time).

;Dependency specification with explicitly defining the delay (delay value 1 means immediately after)
Happens(SOURCE_EVENT, time) → Happens(DEPENDENT_EVENT, time+DELAY).
Happens(DEPENDENT_EVENT, time) → Happens(SOURCE_EVENT, time-DELAY).

In order to use the above pattern for dependency specification, we first need to choose between two different patterns for dependency specification. The first pattern is for dependency specification without explicitly defining the delay between the components, Figure-5.1. As a result the dependent component is started/invoked after some time (which is not explicitly specified) after the completion of other component. Then the second pattern for dependency specification in the model above, is for explicitly specifying the delay with a delay value of 1 signifying immediately after. Then we need to update the source and target events/fluents in the patterns above. For the first pattern the *DEPENDENT_EVENT* should be substituted by the start/invocation event for the dependent component while the *SOURCE_FLUENT* should be substituted by the fluent representing the completion/response reception of the component on which other is dependent. For the second dependency pattern, *SOURCE_EVENT* specifies the completion/invocation end event for the component on which other is dependent, while the *DELAY* specifies the invocation delay of dependent component after the completion of other component.

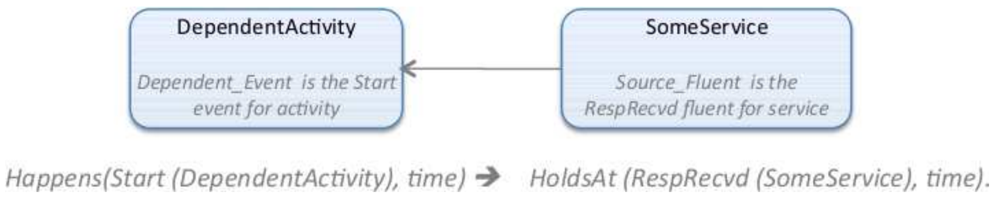

DependentActivity
Dependent_Event is the Start event for activity
SomeService
Source_Fluent is the RespRecvd fluent for service
Happens(Start (DependentActivity), time) ➔ HoldsAt (RespRecvd (SomeService), time).

Figure 5.2: The dependency construct - example

Before going further, let us briefly present the dependency pattern usage for specifying that some activity, *DependentActivity*, is dependent on the completion (response data received) of a service called *SomeService*. An intuitive representation of dependency construct for this particular example is shown in Figure-5.2 and the complete event-calculus axioms are shown below:

Dependency specification patterns - Usage example

;Dependency specification without explicitly defining the delay
Happens(Start_ activitywithoutstate(DependentActivity), time) → HoldsAt (ResponseReceived_ synchservice (SomeService), time).
Happens(Invoke_ synchservice(DependentService), time) → HoldsAt (Finished_ activitywithoutstate (SomeActivity), time).

;Dependency specification with explicitly defining the delay (delay value 1 means immediately after)
Happens(End_ activitywithstate(SomeActivity), time) → Happens (Invoke_ synchservice (DependentService), time+1).
Happens(Invoke_ synchservice(DependentService), time) → Happens (End_ activitywithstate (SomeActivity), time-1).

One important advantage of the proposed approach is that it allows to define events-based dependency and thus dependency can be specified between any two events. This in-turn, makes it possible to specify dependency not only on the successful completion of a component but also on the partial state of a component. Examples include to define the dependency on a activity being started, in execution or on a service being invoked (and not yet completed) or on the data, such as data has been received, expired or the reception of some particular data values.

5.2 Split and Join

The split construct requires the parallel start/invocation of multiple components (termed as split target) after the completion of a component (called split source), Figure-5.3. The split construct can take three forms; the *AND-Split* requires the parallel start/invocation of *all* components, the *OR-Split* requires the parallel start/invocation of *at-least-one* of the components while the *XOR-Split* requires the parallel start/invocation of *exactly-one* of the components specified in the split target.

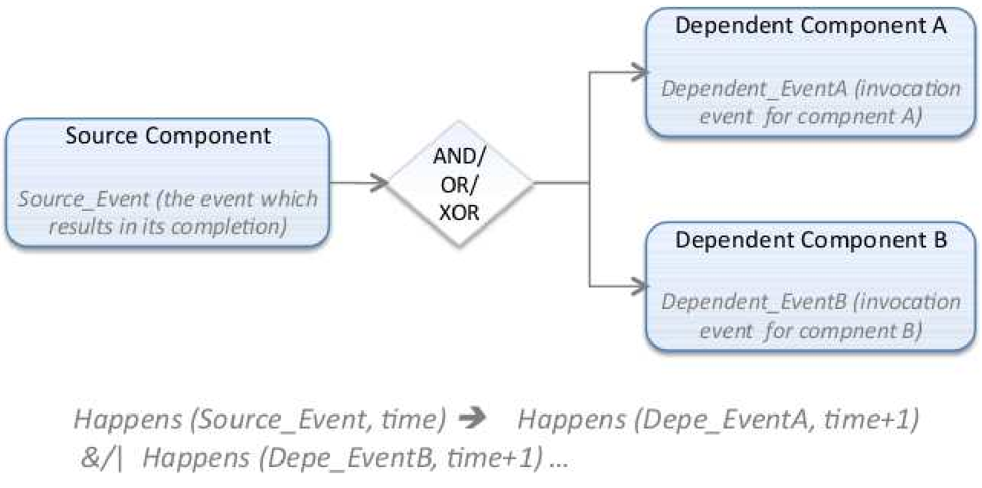

Dependent Component A
Dependent_EventA (invocation event for compnent A)
Source Component
Source_Event (the event which results in its completion)
AND/ OR/ XOR
Dependent Component B
Dependent_EventB (invocation event for compnent B)
Happens (Source_Event, time) ➔ Happens (Depe_EventA, time+1)
&/| Happens (Depe_EventB, time+1) ...

Figure 5.3: The Split construct

An intuitive graphical representation of the structure of event-calculus axioms for specifying the split-construct is shown in Figure-5.3 and below we present the complete event-calculus based patterns for the specification of the split construct with different modes:

Pattern for specifying split construct with different modes

;the AND-Split
Happens(SOURCE_EVENT, time) → Happens(DEPENDENT_EVENT_1, time+DELAY) & Happens(DEPENDENT_EVENT_2, time+DELAY) & ...
Happens(DEPENDENT_EVENT_1, time) | Happens(DEPENDENT_EVENT_2, time) | ... → Happens(SOURCE_EVENT, time-DELAY).

;the OR-Split
Happens(SOURCE_EVENT, time) → Happens(DEPENDENT_EVENT_1, time+DELAY) | Happens(DEPENDENT_EVENT_2, time+DELAY) | ...
Happens(DEPENDENT_EVENT_1, time) | Happens(DEPENDENT_EVENT_2, time) | ... → Happens(SOURCE_EVENT, time-DELAY).

;the XOR-Split
Happens(SOURCE_EVENT, time) → Happens(DEPENDENT_EVENT_1, time+DELAY) | Happens(DEPENDENT_EVENT_2, time+DELAY) | ...
Happens(DEPENDENT_EVENT_1, time) | Happens(DEPENDENT_EVENT_2, time) | ... → Happens(SOURCE_EVENT, time-DELAY).
Happens(DEPENDENT_EVENT_1, time) → !Happens(DEPENDENT_EVENT_2, time) & ...
Happens(DEPENDENT_EVENT_2, time) → !Happens(DEPENDENT_EVENT_1, time) & ...

The event-calculus patterns for split construct presented above, is for the *control-split* however in general, the *split-decision* can also be based on data values or the occurrence of some event. In order to model the event-calculus patterns for data-values based split we need to have notions for conditions that can be evaluated to true are false. The conditions construct is defined later in Section-5.3 and we leave the discussion for data-based split scheme to Section-5.3. The split decision can also be based on the occurrence of some event and below we discuss the event-calculus patterns for event-based split scheme:

Pattern for specifying event-based (AND/OR)split construct

Happens(SOURCE_EVENT, time) & Happens(CONDITIONAL_EVENT, time) → Happens(DEPENDENT_EVENT_1, time+DELAY) (&,|) Happens(DEPENDENT_EVENT_2, time+DELAY) (&,|) ...

Happens(DEPENDENT_EVENT_1, time) | Happens(DEPENDENT_EVENT_2, time) | ... → Happens(SOURCE_EVENT, time-DELAY) & Happens(CONDITIONAL_EVENT, time-DELAY).

In the model above, we update the patterns for specifying the split construct and added the axiom to split iff *CONDITIONAL_EVENT* happens. As discussed earlier, one important aspect of the proposed approach is that it allows to define constructs (such as the split construct) not only on the successful completion of a component but also on the partial state of a component. In general, the constructs such as split are thus defined in terms of events and any specified target events can happen in parallel after the occurrence of the source event.

Further, the join construct handles the control aggregation after the parallel invocation of components using the split construct discussed above and requires the components (to which control was splitted) to complete before the start/invoke of the component after the join construct. Different aggregation schemes can be used that can be used requiring *all/exactly-one/at-least-one/subset* of components to complete. The join construct can be modeled using event-calculus by defining dependency (using the patterns for the dependency construct defined earlier) for the component following the join construct on (*all/exactly-one/at-least-one/subset-of*) components to which control was routed using the split construct.

5.3 Conditions

In order to handle the conditional invocation of components, we introduce the *condition* construct that signifies a data-based condition that can be evaluated to be either true or false and other events can be based on conditions evaluation. In order to model conditions we can define an event-calculus sort called *condition*, whose instances represent the conditions to be evaluated. Then to abstract the condition evaluation process, we can define two events called *EvaluateCondition-True(condition)* and *EvaluateConditionFalse(condition)*, and the corresponding fluents named *ConditionTrue(condition)* and *ConditionFalse(condition)*, that represent if the condition is evaluated to *true* or *false*. Further, we add an axiom that specifies, either the condition is evaluated to *true* or *false* and not both. The event-calculus model below is used for defining conditions:

Conditions specification using event-calculus - *condition.e*
Usage: import condition.e and define instances of sort condition to represent conditions

```
sort condition

event EvalConditionTrue(condition)
event EvalConditionFalse(condition)

fluent ConditionTrue(condition)
fluent ConditionFalse(condition)

Initiates(EvalConditionTrue(condition), ConditionTrue(condition),time).
Initiates(EvalConditionFalse(condition), ConditionFalse(condition),time).

HoldsAt(ConditionTrue(condition), time) → !Happens(EvalConditionTrue(condition), time) &
!Happens(EvalConditionFalse(condition), time).
HoldsAt(ConditionFalse(condition), time) → !Happens(EvalConditionFalse(condition), time) &
!Happens(EvalConditionTrue(condition), time).

Happens(EvalConditionTrue(condition), time) → !Happens(EvalConditionFalse(condition), time).
Happens(EvalConditionFalse(condition), time) → !Happens(EvalConditionTrue(condition), time).

!HoldsAt(ConditionTrue(condition), 0).
!HoldsAt(ConditionFalse(condition), 0).
```

The conditions model defined above can then also be used to define data-based split scheme mentioned in Section-5.2 where the decision to split the control should be based on the evaluation of some particular condition. An intuitive graphical representation of the structure of event-calculus axioms for specifying the data-based split scheme is shown in Figure-5.4.

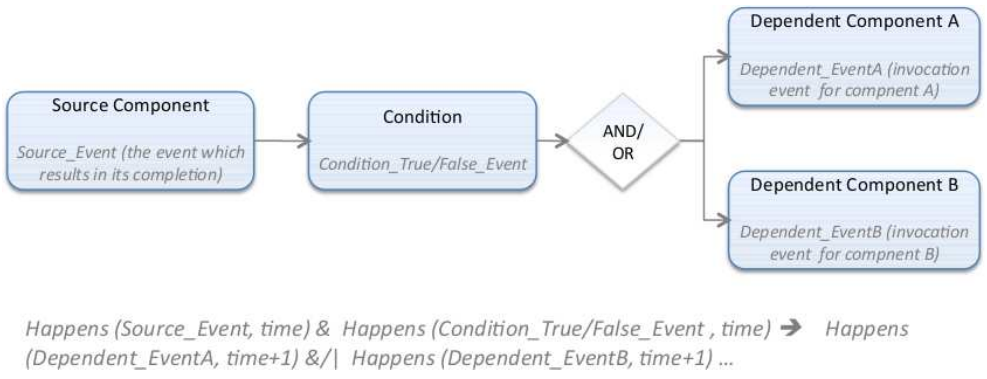

Source Component
Source_Event (the event which results in its completion)
Condition
Condition_True/False_Event
AND/ OR
Dependent Component A
Dependent_EventA (invocation event for compnent A)
Dependent Component B
Dependent_EventB (invocation event for compnent B)
Happens (Source_Event, time) & Happens (Condition_True/False_Event , time) ➔ Happens (Dependent_EventA, time+1) &/| Happens (Dependent_EventB, time+1) ...

Figure 5.4: The data-based split construct

In the model below, we update the patterns for specifying the split construct and add the axiom to split iff *CONDITION_ TRUE_ EVENT* (or condition false event, if needed) happens.

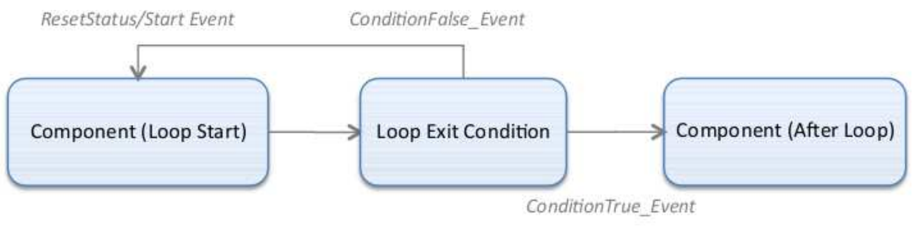
Pattern for specifying data-based (AND/OR/XOR)split construct

Happens(SOURCE_EVENT, time) & Happens(CONDITION_TRUE_EVENT, time) → Happens(DEPENDENT_EVENT_1, time+DELAY) (&,|) Happens(DEPENDENT_EVENT_2, time+DELAY) (&,|) ...

Happens(DEPENDENT_EVENT_1, time) | Happens(DEPENDENT_EVENT_2, time) | ... → Happens(SOURCE_EVENT, time-DELAY) & Happens(CONDITION_TRUE_EVENT, time-DELAY).

5.4 Iteration

Iteration construct requires a set of components (termed as *loop-body*) to be invoked continuously until some condition (termed as *loop-exit-condition*) does not holds. In order to model the iteration construct using the proposed framework, we can first define dependency (using the dependency construct discussed earlier) amongst the components included in the loop-body and finally specifying the dependency for the loop-exit-condition on the last component included in the loop body. Further, we can define an event-calculus axiom to either exit the loop (if the condition holds) or to re-start the first component in the loop-body if the condition does not hold, Figure-5.5.



Happens (ConditionFalse_Event, time) ➔ Happens (ResetStatus_Event_LoopStartComp, time) & Happens (ResetStatus_Event_LoopBodyComp, time) ...
Happens (ConditionTrue_Event, time) ➔ Happens (Start_Event_LoopStartComp, time+1).

Happens (ConditionFalse_Event, time) ➔ Happens (Start_Event_AfterLoopComp, time+1).

Figure 5.5: Patterns for specifying Iteration construct

One important requirement for the iteration construct is that all the components (and conditions including loop exit condition) included in the loop-body should be possible to be restarted/reinvoked (and thus the event-calculus models added to the process specification, should be appropriate ones). Further, once the loop-exit-condition does not hold and requiring iteration, the status of all the components should be reset. The event-calculus model below specifies the patterns for specifying iteration construct:

Pattern for specifying iteration construct

Happens(LOOP_EXIT_CONDITION_FALSE_EVENT, time) → Happens(START_EVENT, time+DELAY)
Happens(LOOP_EXIT_CONDITION_FALSE_EVENT, time) → Happens(RESET_EVENT_1, time+DELAY) & Happens(RESET_EVENT_2, time+DELAY) ...

Happens(RESET_EVENT_1, time) | Happens(RESET_EVENT_2, time) ... → Happens(LOOP_EXIT_CONDITION_FALSE_EVENT, time-DELAY)

5.5 Request/Response data

The Web services and activities models presented earlier in this chapter do not cater for request and response data (and their properties) that are associated with different components, for instance input/output parameters of some service. In order to model request and response data we introduce new sorts named *requestdata* and *responsedata*, respectively. The instances of these sorts then represent the request and response data elements.

Then, in order to associate these data elements to the components they belong, for instance specifying the *search_query* input data element (instance of sort *requestdata*) belongs to the *Search* synchronous Web service (instance of sort *synchservice*) we have explored two approaches. First, we can define a predicate called *HasInput(synchservice, requestdata)* and we can specify the predicate for individual instances, *HasInput(Search, search_query)*, and then we can use this predicate in axioms. However, we believe that this approach results in complex axioms which may increase the problem size (and the time taken for event-calculus to SAT encoding process). The other approach requires to specify some data based properties (such as data *availability*, *validity*, *confidentiality* and others) in form of event-calculus axioms and fluents that need to be evaluated for the request and response data. Then, we can associate these data-based events and fluents to the service/activity based events. Below we present the model for the request data (response data model is very much similar):

Request data for Web services *- requestdata.e*
Usage: *- import requestdata.e and define instances of sort requestdata to represent request data*

```
sort requestdata

event Validate_requestdata(requestdata)
event Encrypt_requestdata(requestdata)
fluent IsValid_requestdata(requestdata)
fluent IsEncrypted_requestdata(requestdata)
Initiates(Validate_requestdata(requestdata), IsValid_requestdata(requestdata),time).
Initiates(Encrypt_requestdata(requestdata), IsEncrypted_requestdata(requestdata),time).

HoldsAt(IsValid_requestdata(requestdata), time) → !Happens(Validate_requestdata (request-
data),time).
HoldsAt(IsEncrypted_requestdata(requestdata), time) → !Happens(Encrypt_requestdata (request-
data),time).

!HoldsAt(IsEncrypted_requestdata(requestdata),0).
!HoldsAt(IsValid_requestdata(requestdata),0).
```

In order to use the model we need to import the model to the specification and then define instances of the sort *requestdata*. Further, we need to specify when the properties are evaluated for each data element, for instance the request data associated with a service needs to be encrypted before the request is to be made.

5.6 Message flow

In order to model the message flow between different components, we can also define an event-calculus sort called messagedata and its instances represent the data elements that need to be transferred between components. Then each *messagedata* has a source component, from which the data being sent, and a target component which is the receiver of the data. These can be abstracted by defining events *Send_messagedataReceive_messagedata* and corresponding fluents *Sent_messagedataReceived_messagedata*. Finally we can define axioms to specify that the *Send_messagedata* event happens when the source component starts (or is in any intermediate state) and the *Receive_messagedata* event happens when the target component starts (or is in any intermediate state). The event-calculus model below handles message flow between components (*messagedata* abbreviated as *msgdata*):

***Message-flow specification using event-calculus** - messagedata.e*
Usage: import messagedata.e and create instances of sort messagedata

```
sort msgdata

event Validate_msgdata(msgdata)
event Send_msgdata(msgdata)
event Receive_msgdata(msgdata)

fluent IsValid_msgdata(msgdata)
fluent Sent_msgdata(msgdata)
fluent Received_msgdata(msgdata)

Initiates(Validate_msgdata(msgdata), IsValid_msgdata(msgdata),time).
Initiates(Send_msgdata(msgdata), Sent_msgdata(msgdata),time).
Initiates(Receive_msgdata(msgdata), Received_msgdata(msgdata),time).

HoldsAt(IsValid_msgdata(msgdata),time) → !Happens(Validate_msgdata(msgdata),time).
HoldsAt(Sent_msgdata(msgdata), time) → !Happens(Send_msgdata(msgdata),time).
HoldsAt(Received_msgdata(msgdata), time) → !Happens(Receive_msgdata(msgdata),time).
Happens(Validate_msgdata(msgdata),time) → HoldsAt(Received_msgdata(msgdata), time).

!HoldsAt(IsValid_msgdata(msgdata),0).
!HoldsAt(Sent_msgdata(msgdata),0).
!HoldsAt(Received_msgdata(msgdata),0).
```

5.7 Example

We will now review the motivating example (Section-2.2) and discuss event-calculus modeling for different control/data flow aspects related to the example. As discussed earlier in Section-2.3.5, we will consider process fragments modeling from an individual component perspective and later in Chapter-7 we will discuss process instantiation that aims to find a solution by connecting these fragments. We identified different components for the motivating example in Section-4.5 and as the invocation of different components depend on some conditions to be evaluated, we also add some conditions to the process specification as shown in the model below:

Motivating example - Defining conditions

```
...
load includes/condition.e
condition Is(CRR/SC/FC)Available, Has(CRR/SC/FC/FB)Arrived, FalseAlarmCheckByCRR,
FalseAlarmCheckByFB, HasConservatorArrived, IsExternalHelpNeeded
```

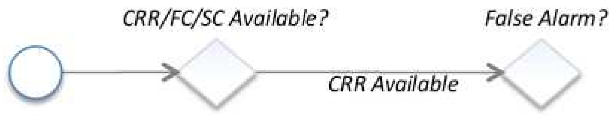


Figure 5.6: Motivating example - Process start fragment

Regrading the control/data flow requirements for the motivating example, we first specify that at the start of the process we need to first check if the

CRR/FC/SC are available, Figure-5.6. The first axiom in the model below is for checking CRR availability and we have omitted the axioms for FC/SC availability as they are very similar. Further, if CRR is available at the site he can check that if it is not the false alarm, the second axioms. In the model below we will abbreviate *Start_activitywithoutstate* as *Start_AWS* and *EvalCondition(True/False)* as *EvalCond(True/False)*.

Motivating example - Checking if CRR is available, if so check for the FalseAlarm

;at start check condition
Happens(Start_AWS(Start), time) → Happens(EvalCondTrue(IsCRRAvailable), time) | Happens(EvalCondFalse(IsCRRAvailable), time).

;check if it is a false alarm only if CRR is available
Happens(EvalCondTrue (FalseAlarmCheckByCRR), time) | Happens(EvalCondFalse (FalseAlarmCheckByCRR),time) → HoldsAt (ConditionTrue (IsCRRAvailable),time).

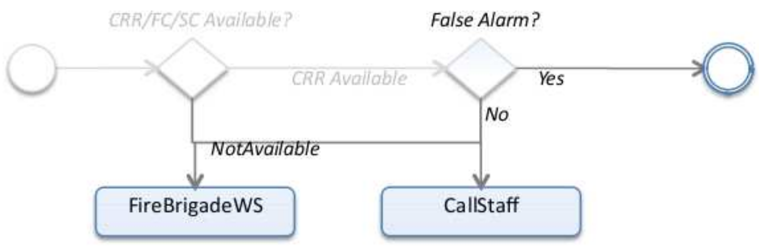


Figure 5.7: Motivating example - Invocation of *FireBrigadeWS* and *CallStaffWS*

Next, in the model below we specify to end the process if CRR is available and he decides that it is a false alarm (the first axiom) and if not the case invoke the *FireBrigadeWS* and *CallStaffWS* Web services, Figure-5.7. These services can also be invoked automatically if the CRR is not there and the last two axioms in the model below handle this requirement.

Motivating example - If FalseAlarm end process, otherwise invoke services

;if indeed it is a false alarm then exit
Happens(Start_AWS(End1), time) → HoldsAt(ConditionTrue (FalseAlarmCheckByCRR),time).

;two cases for the invocation of FireBrigadeWS
Happens(Invoke_synchservice(FireBrigadeWS), time) → HoldsAt(ConditionFalse (FalseAlarmCheckByCRR),time) | HoldsAt (ConditionFalse(IsCRRAvailable),time).

;two cases for the invocation of CallStaffWS
Happens(Invoke_synchservice(CallStaffWS), time) → HoldsAt(ConditionFalse (FalseAlarmCheckByCRR),time) | HoldsAt (ConditionFalse(IsCRRAvailable),time).

Then, the CRR if available can invoke other services as well but as we are discussing event-calculus models from individual components point of view and in terms of security requirements for the process CRR should always be available for invoking other emergency services. There are two cases, is he already available as modeled above or either he is not there and once the *CallStaffWS* service is invoked he is there after some delay, Figure-5.8. In the model below we handle the arrival of CRR once the *CallStaffWS* is invoked and only if he is not already there at the site.

Motivating example - If CRR not available, he is there after CallStaffWS is invoked

Happens(EvalCondTrue(HasCRRArrived), time) | Happens(EvalCondFalse(HasCRRArrived), time) → HoldsAt(ResRecvd_synchservice (CallStaffWS),time) & !HoldsAt(ConditionTrue (IsCRRAvailable),time).

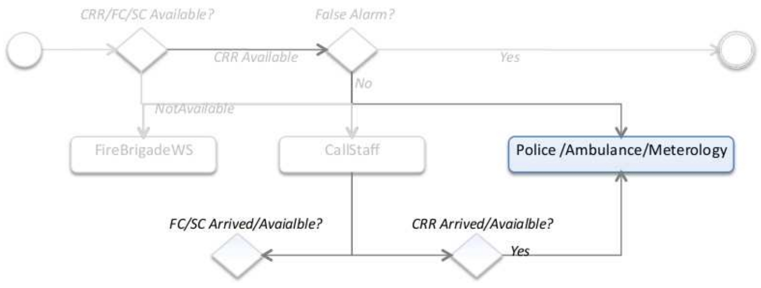


Figure 5.8: Motivating example - Invocation of other emergency services

We can have similar model as shown above for SC and FC. Then, once CRR has reached the site (or was already there) he can invoke other emergency services, Figure-5.8, as modeled below.

Motivating example - Other emergency services invocation

Happens(Invoke_synchservice(PoliceWS), time) → HoldsAt(ConditionFalse (FalseAlarmCheckByCRR),time) | HoldsAt(ConditionTrue (HasCRRArrived),time).

Happens(Invoke_synchservice(AmbulanceWS), time) → HoldsAt(ConditionFalse(FalseAlarmCheckByCRR),time) | HoldsAt(ConditionTrue (HasCRRArrived),time).

Happens(Invoke_synchservice(MeteorologyDeptWS), time) → HoldsAt(ConditionFalse (FalseAlarmCheckByCRR),time) | HoldsAt(ConditionTrue (HasCRRArrived),time).

Now moving to other part of the process, Figure-5.9, the fire-brigade arrives after the *FireBrigadeWS* is invoked and the condition that *HasFBArrived* is thus evaluated after the request to the *FireBrigadeWS* has been made. The first axiom in the model below handles this behavior and once the fire brigade arrives they can check also check if it is just a false alarm (or the scale of fire does not need containment), if so the process ends. The last two axioms in the model below handle this requirement:

Motivating example - Fire-brigade arrival and checking if fire indeed needs containment

Happens(EvalCondTrue(HasFBArrived), time) | Happens(EvalCondFalse(HasFBArrived), time) → HoldsAt(RespRecvd_ synchservice(FireBrigadeWS),time).

Happens(EvalCondTrue(FalseAlarmCheckByFB), time) | Happens(EvalCondFalse (FalseAlarm-CheckByFB), time) → HoldsAt(ConditionTrue(HasFBArrived),time).

Happens(Start_ activitywithoutstate(End2), time) → HoldsAt(ConditionTrue (FalseAlarmCheck-ByFB),time).

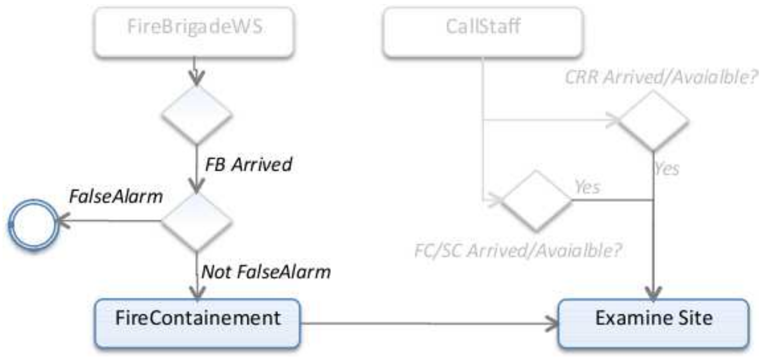


Figure 5.9: Motivating example - *FireContainement* and *ExamineSite* activities

Further, once the fire-brigade arrives and they decide it is not a false alarm, the fire containment starts, the first axiom in the model below handle this requirement. Then, once the fire-containment finishes and once the CRR has arrived (or was already there), examine site activity starts. The last axiom in the model below handle this requirement, *Start/End_ activitywithstate* abbreviated as *Start/End_ activityWS*).

Motivating example - Fire-containment and ExamineSite

Happens(Start_ activityWS(FireContainment), time) → HoldsAt(ConditionFalse (FalseAlarm-CheckByFB),time).

Happens(Start_ activityWS(ExamineSite), time) → HoldsAt(Finished_ activitywithstate (FireContainment), time) & (HoldsAt(ConditionTrue (HasCRRArrived),time) | HoldsAt(ConditionTrue (Is-CRRAvailable),time)).

Once the site has been examined, it is decided whether external help is needed and if so, the *ExternalOrgWS* is invoked (the first two axioms in the model below). Then, the activity to plan recovery can only be started if the decision on external help is taken and once CRR, SC and FC are all available at the site (either they were already there at the site or have reached the site once the *CallStaffWS* has been invoked), Figure-5.10. The last axiom (only shown for CRR but also contains conditions for SC/FC) in the event-calculus model below handles this requirement.

Motivating example - External help decision and Plan recovery activity

Happens(EvalCondTrue(IsExternalHelpNeeded), time) | Happens(EvalCondFalse (IsExternalHelpNeeded), time) → HoldsAt(Finished_ activityWS(ExamineSite), time).

Happens(Invoke_ synchservice(ExternalOrgWS), time) → HoldsAt(ConditionTrue (IsExternalHelpNeeded),time).

Happens(Start_ activityWS(PlanRecovery), time) → (HoldsAt(RespRecvd_ synchservice (ExternalOrgWS), time) | HoldsAt(ConditionFalse(IsExternalHelpNeeded),time)) & (HoldsAt(ConditionTrue(HasCRRArrived),time) | HoldsAt(ConditionTrue(IsCRRAvailable),time)) & ... conditions for SC/FC.

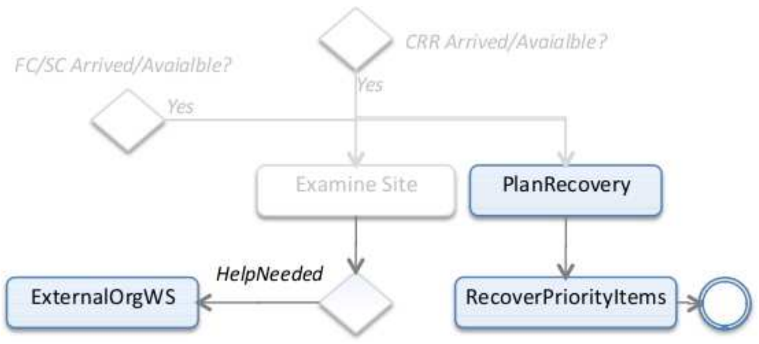


Figure 5.10: Motivating example - *PlanRecovery* and *RecoverPriorityItems* activities

Finally the activity for the recovery of the priority items (*RecoverPriorityItems*) can start once the planning has finished, Figure-5.10. In the model below, we also specify the composition goal that is to have the items recovered 90 minutes after the start of the process.

Motivating example - Recover items activity and the composition goal

Happens(Start_ activityWS(RecoverPriorityItems),time) → HoldsAt(Finished_ activityWS (PlanRecovery),time).

; composition goal
HoldsAt(Finished_ activitywithstate(RecoverPriorityItems), 90).

5.8 Summary

In this chapter we have discussed the event-calculus 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. The table below provides an overview regarding the structure of this chapter.

Construct	***Section for event-calculus model***
Dependency	*Event-calculus based patterns for specifying dependency between two components is discussed in Section-5.1.*
Split & Join	*Patterns for Split and Join constructs (including different split/join schemes) is presented in Section-5.2.*
Conditions	*Event-calculus model for specifying conditions is discussed in Section-5.3 and corresponding include file is condition.e.*
Iteration	*Patterns for specifying iteration construct is discussed in Section-5.4.*
Data	*Event-calculus model for specifying request/response data is discussed in Section-5.5, corresponding files are request.e and response.e.*
Messages	*EC model for specifying the message transfer between two components including patterns for its usage is discussed in Section-5.6, corresponding include file is messagedata.e.*

We have also discussed the control/data flow specification for the motivating example, Section-2.2, and discussed how the proposed patterns can be used to specify the control/data flow between different components identified for the motivating example. This pattern based approach has allowed us to implement a Java-based application, called *ECWS*, that can automatically generate the event-calculus models for the process specification, we will discuss the *ECWS* application in Section-9.2.

Chapter 6

Modeling Non-functional aspects

Contents

In this chapter we will discuss how the proposed approach allows to model different security and temporal aspects related to the composition process. We will first discuss patterns for specifying temporal requirements and then will discuss how the event-calculus models for specifying different security requirements for the composition process.

6.1 Modeling temporal aspects

Using event-calculus as the modeling formalism also allows to specify the temporal constraints for the composition process. The temporal constraints can either be local to a component (such as specifying the time it takes for a service to produce response) or they can be imposed by the composition process on the participating components (such as specifying the delay between the invocation of two services). Further, the temporal constraints can be either control based (such as specifying the delay between the successive invocation of component) or they can be data-based temporal constraints (such as data validity constraints).

6.1.1 Response time

The response time temporal constraint specifies the time it takes for a component to finish execution, such as the time taken for Web service to produce response once a request is made or the time it takes for an activity to be completed. The choice of event-based approach allows to specify delay between events and for instance, makes it possible to specify the time it takes for an activity to get started/finished or the time spent during execution. The response time is a local constraint and is specified by the participating components requiring the composition process to cater for this constraint in finding the solution to the composition process. The event-calculus pattern below can be used to specify the response-time constraint:

Pattern for specifying response-time temporal constraint

Happens(START_EVENT, time) → Happens(END_EVENT, time+RESPONSE_DELAY).
Happens(END_EVENT, time) → Happens(START_EVENT, time-RESPONSE_DELAY).

In the model above, the *START_EVENT* represents the component start event (such as) and the *END_EVENT* represents the end event for the same component. Finally the *RESPONSE_DELAY* represents the response-time delay for the component.

6.1.2 Restart/Refresh

The temporal constraints also include the restart constraint which requires a component to be restarted/reinovked after a fixed time, for instance re-invoking some backup service after a fixed delay. The restart constraint when applied to data flow, requires the data to be re-fetched (by re-invoking service/task) once it expires and is called data refresh constraint. Below we present the patterns for specifying the restart/refresh temporal constraint using event-calculus.

Pattern for specifying restart/refresh temporal constraint

Happens(SOURCE_EVENT, time) → Happens(RESTART_EVENT, time+DELAY).
Happens(DATA_EXPIRY_EVENT, time) → Happens(REFRESH_EVENT, time+DELAY).

In the model above, the first pattern is for specifying the restart constraint while the other one is for the data refresh temporal constraint. For the restart constraint, the *SOURCE_EVENT* represents the event after which the component should be restarted (for instance, some condition RequiresBackup being true). Then for the refresh constraint the *DATA_EXPIRY_EVENT* represents the data expiry event after which the component should be restarted to fetch the data. One important requirement for the restart/refresh constraint is that the component should support restart and its status has been reset once the *SOURCE_EVENT* or the *DATA_EXPIRY_EVENT* happens.

6.1.3 Invocation time-frame and delay

The invocation timeframe constraint requires a service to be invoked (task to be started) within a fixed timeframe. The specified time-frame can either be absolute time or the time that has been passed after any particular event and serves as a boundary for the times in which service/task can be invoked. As an example, consider the service to backup records should start between some specific time interval or it can be specified that the backup service should be invoked after the data has been updated. Further it is also possible to specify the exact time at which the service/task should be invoked. In terms of data flow, the timeframe constraint requires that the some particular data data is available (at-least/only) between the specified time-frame. Below we model different types of invocation time frame constraints; the first axiom models the control flow based invocation time-frame constraint. Then, the other axiom model the component invocation/start at exact time-point:

Pattern for specifying invocation time-frame and delay temporal constraints

Happens(COMPONENT_START_EVENT,time) → time>IntervalStartTime & time<IntervalEndTime.
Happens(SOME_EVENT, SomeExactTimepoint).

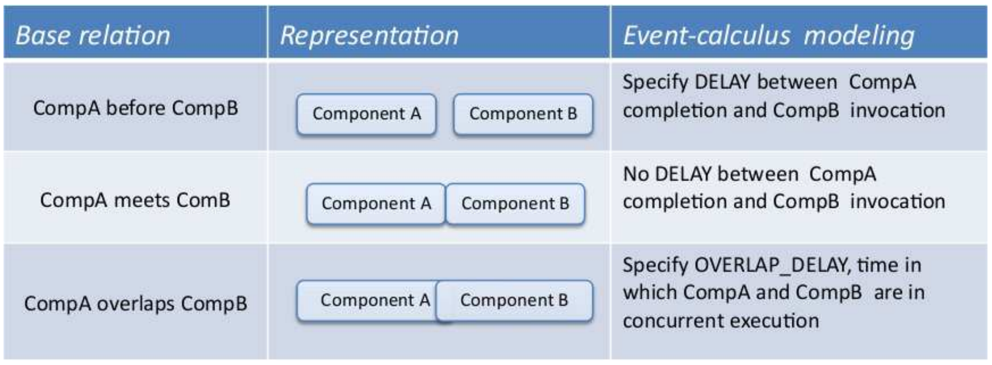
The execution delay constraint requires that the successive invocations of the service must be delayed by some time (possibly to prevent overloading a service). This constraint can also be specified to specify the invocation delay between multiple services, such as some backup service must be invoked some time after the service that has changed the data.

6.1.4 Allen's Interval Algebra

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[1].

The base relations for the Allen's interval algebra can be mapped and applied to the proposed framework and below we discuss the patters for specifying these relations using event-calculus:

[1]http://en.wikipedia.org/wiki/Allen's_Interval_Algebra



Base relation	Representation	Event-calculus modeling
CompA before CompB	Component A Component B	Specify DELAY between CompA completion and CompB invocation
CompA meets ComB	Component A Component B	No DELAY between CompA completion and CompB invocation
CompA overlaps CompB	Component A Component B	Specify OVERLAP_DELAY, time in which CompA and CompB are in concurrent execution

Figure 6.1: Base relations (*after, meets, overlaps*) for Allen's Interval Algebra

Patterns for specifying base relations of Allen's interval algebra
Usage:

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).

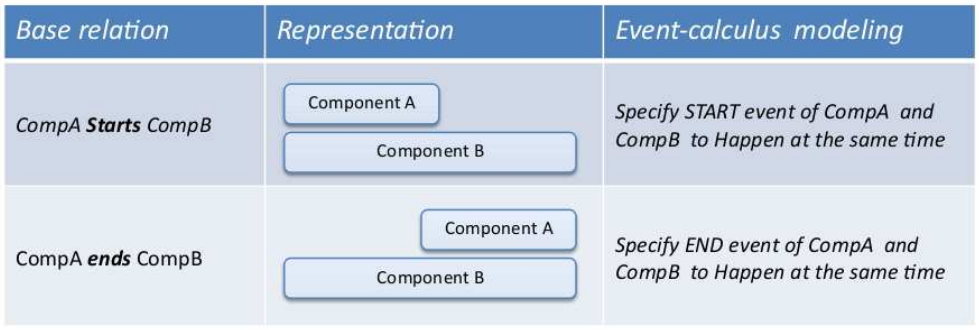
The patterns above are for *after/meets/overlaps* relations proposed in the Allen's interval algebra, Figure-6.1. *EVENT_A* is the start event of component *COMP_A* and *EVENT_B* is the end event of the component *COMP_B*. For the after relation the *DELAY* specifies the delay time between the start of the components and for the meets relation there is no *DELAY*, as *COMP_A* starts at the same time as *COMP_B* finishes. Further, in case of the overlaps relation, the delay is called the *OVERLAP_DELAY* which specifies the time in which both components are in concurrent execution.

Patterns for specifying base relations of Allen's interval algebra
Usage:

COMP_A starts COMP_B
Happens(EVENT_A, time) → Happens(EVENT_B, time).

COMP_A ends COMP_B
Happens(EVENT_A, time) → Happens(EVENT_B, time).

The patterns above are for *starts/ends* relations proposed in the Allen's interval algebra, Figure-6.2. For the starts (ends) relation *EVENT_A* is the start (end) event of component *COMP_A* and *EVENT_B* is the start (end) event of the component *COMP_B*.



Base relation	Representation	Event-calculus modeling
CompA **Starts** CompB	Component A / Component B	Specify START event of CompA and CompB to Happen at the same time
CompA ***ends*** CompB	Component A / Component B	Specify END event of CompA and CompB to Happen at the same time

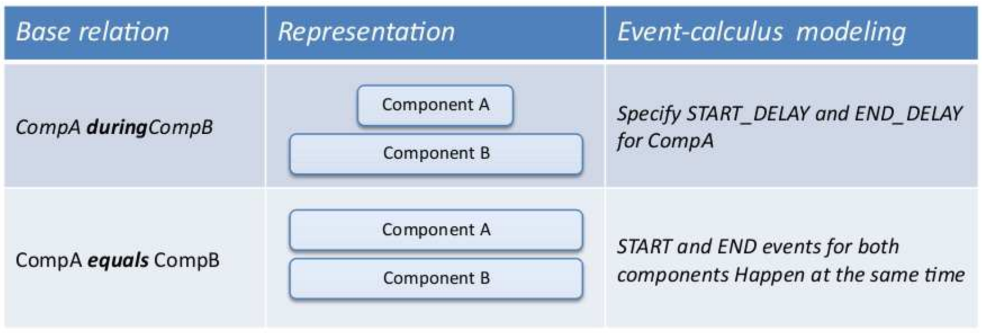
Figure 6.2: Base relations (*starts, ends*) for Allen's Interval Algebra

Patterns for specifying base relations of Allen's interval algebra
Usage:

COMP_A during(starts after and finishes before) COMP_B
Happens(EVENT_B_START, time) → Happens(EVENT_A_START, time+START_DELAY).
Happens(EVENT_A_START, time) → Happens(EVENT_B_START, time+START_DELAY).

Happens(EVENT_B_END, time) → Happens(EVENT_A_END, time-END_DELAY).
Happens(EVENT_A_END, time) → Happens(EVENT_B_END, time+END_DELAY).

The patterns above are for *during* and *equals* relations proposed in the Allen's interval algebra, Figure-6.3. $EVENT\_A\_START$ and $EVENT\_A\_END$ are the start/end events of the component $COMP\_A$ while $EVENT\_B\_START$ and $EVENT\_B\_END$ are start/end events of the component $COMP\_B$. For the during relation, the $START\_DELAY$ specifies after how much time the $COMP\_A$ should start after starting $COMP\_B$ and END_DELAY specifies before how much time is should finish before $COMP\_B$ finishes. The equals relation uses the same axioms as for the during relation, without specifying the delay (or having delay equals 0).



Base relation	Representation	Event-calculus modeling
CompA **during**CompB	Component A / Component B	Specify START_DELAY and END_DELAY for CompA
CompA ***equals*** CompB	Component A / Component B	START and END events for both components Happen at the same time

Figure 6.3: Base relations (*during, equals*) for Allen's Interval Algebra

6.1.5 Modeling time-units

The event-calculus models presented above discuss the events that happens and fluents that hold at a particular time-point. However, in order to model the Web services with temporal constraints in different time-units, as synchronous services take seconds while the asynchronous can take minutes and longer, we need to add semantics to the event-calculus time-points. This will allow us to treat time-points as the actual time and thus specifying that an event happens at event-calculus time-point 1, can signify that the event happens at one second/minute and so on. The possible solutions to this problem include to convert all the time-units to some common time-unit (such as seconds), however in reference to proposed implementation architecture which attempt to convert the composition process to a SAT based problem and then invokes a SAT based solver, this choice is unfeasible. A smaller common time-unit such as seconds, and converting all other units to seconds (10 minutes means 600 seconds) will increase the resulting SAT encoding size and thus is not feasible. On the other hand, converting all the units to a higher common format such as minutes will not allow to reason about the smaller time units, such as seconds, as for the implementation DECReasoner is discrete.

As an alternative, we pre-process the time-units associated with different participating services to first convert all the time-units to a common format such as seconds. Then, an order is sought based on the time they take and finally that order is used to redefine the time they take. A solution is then sought with the updated constraints using the reasoner and the solution is then post-processed to update the time-units associated with each service.

6.2 Modeling security aspects

The choice of event-calculus as the modeling formalism also allows to handle the security requirements in the composition process. Some initial work on modeling security policies using event-calculus and to reason about them can be found in [Bandara 2003]. In section, we will first briefly discuss some of the security requirements for the composition process and then we will discuss the different levels at which the security aspects can be incorporated in the services composition. Then, we will discuss how the proposed framework allows to model these requirements at different interaction levels.

6.2.1 Security requirements

The different security requirements for the composition process, are shown in the table below[Souza 2009]:

Requirement	*Description*
Authentication	*requires to identify the user requesting a critical resource and thus allowing only the legitimate users have access to the critical information.*
Authorization	*requires to identify what operations the user is allowed to perform and if the user is allowed to access the critical resource they are requesting.*
Confidentiality	*security property requires that the critical information, such as credit card and other personal information, should be encrypted and protected from unauthorized access.*
Data integrity	*security property requires that the sensitive data (such as personal information of a user) has to be verified for data corruption (such as tampering of data between the sender and receiver) before usage.*
Data retention	*property associates a time-to-live (TTL) information with the data requiring it to be deleted after a certain time.*
Auditing	*property requires that all the operations performed by the composition process should be logged and are available for auditing (if needed).*
Non-repudiation	*property requires that any user performing an operation (such as accessing critical information) can not later deny that action and thus the non-repudiation requires support from authentication, integrity and auditing process.*

6.2.2 Interaction levels

The above mentioned security requirements can be handled at different interaction levels for the participating Web services in the composition process. At the transport layer, existing Web tier technology such as SSL can be used to handle security requirements such as SSL encrypted connections for confidentiality and to avoid data interception. Then at the message level, security data can be added to the SOAP header fields to be then handled by the SOAP server. Further, the security requirements can further be handled at the application level by using the application specific encryption, authorization checks and others.
The proposed framework allows to model the security requirements at different interaction levels for Web services. Below we briefly discuss the event-calculus modeling for the different security requirements identified earlier, and for each requirement we will discuss the event-calculus modeling for different interaction levels.

6.2.3 Data confidentiality, retention and integrity

Confidentiality security property requires that the critical information, such as credit card and other personal information, should be encrypted and protected from unauthorized access. At the transport layer, confidentiality can be achieved by using, for instance, SSL encrypted connections. Then at the message level, XML Encryption specification can be used while at the application layer some application level encryption schemes can be used. Using event-calculus, the confidentiality requirement can be modeled at different interaction levels and we discussed earlier the event-calculus models for specifying properties such as encryption for the request/response data. In general, we can define event-calculus events/fluents that abstract the encryption at different levels and then define axioms that specify when a particular encryption level is used. In the event-calculus model below we first attempt to model the transport level data confidentiality and then present the pattern to handle the application-level confidentiality requirement:

Confidentiality - Transport and Application level

event UseEncryptedSSLConnection()
fluent EncryptedSSLConnectionUsed()
Initiates(UseEncryptedSSLConnection(), EncryptedSSLConnectionUsed(),time).
!HoldsAt(EncryptedSSLConnectionUsed(), 0).

Happens(COMPONENT_START_EVENT, time) $\rightarrow$ *HoldsAt(IsEncrypted_requestdata (SOME_DATA_ELEMENT), time).*

Before going further, let us briefly discuss how the proposed modeling approach can be used to reason about the confidentiality requirement. First, as the confidentiality can b handled at different levels and an application may opt to choose one particular encryption level and thus not opting for encryption levels. Then, for the application level encryption we can have axioms that the data should not be used once decrypted and should not be transferred to other services before encryption. Further, we can use the reasoner to perform a compatibility analysis of services that support different encryption schemes, for instance if a service is using some particular application level encryption scheme then we need to identify if some other service consuming data from that service can decrypt the data.

The **data retention** requirement associates a time-to-live (TTL) information with the data requiring it to be deleted after a certain time and we earlier modeled the request/response data that can be expired. The data retention property can thus be defined as defining the axioms that specify the delay between the request/response messages and their expiry. The event-calculus model below presents the pattern to be used to specify the data invalidity (responsedata) in this case after some event.

Data Retention

Happens(SOME_EVENT, time) $\rightarrow$ *Happens(Invalidate_responsedata(SOME_DATA_ELEMENT), time).*

Further, the **data integrity** property requires that the sensitive data (such as personal information of a user) has to be verified for data corruption (such as tampering of data between the sender and receiver) before usage. Using event-calculus, the data integrity requirement can be modeled at different interaction levels and we discussed earlier the event-calculus models for specifying properties such as checking data validity for the request/response data and they serve as an example for the application-level data integrity model. In general, we can define event-calculus events/fluents the data validation at different levels and then define axioms that specify when a particular validation is performed.

6.2.4 Authentication/Authorization

The ***authentication*** property requires that only appropriate users have access to the sensitive or critical information held by the services while the ***authorization access control*** restricts the access to resources to only the authorized users. As similar to the event-calculus models for confidentiality and integrity, we can define events and fluents to model the authentication and authorization at different interaction levels, as shown below:

Access control
Usage: Specify users by defining instances of sort user, then use axioms to specify the access required by each event.

```
sort user
event AuthorizeUser(user)
event RevokeAuthorization(user)
fluent UserAuthorized(user)

Initiates(AuthorizeUser(user), UserAuthorized(user), time).
Terminates(RevokeAuthorization(user), UserAuthorized(user), time).

Happens(RevokeAuthorization(user), time) → HoldsAt(UserAuthorized(user), time).
HoldsAt(UserAuthorized(user), time) → !Happens(AuthorizeUser(user),time).

!HoldsAt(UserAuthorized(user),0).
```

In order to use the authorization model mentioned above, we can use a set of patterns as below.

Access control usage patterns

```
Happens(SOME_EVENT, time) → Happens(AuthorizeUser(SOME_USER), time-DELAY).
Happens(AuthorizeUser(SOME_USER), time) → Happens(SOME_EVENT, time+DELAY) |
Happens(SOME_OTHER_EVENT, time+DELAY)É list of all events that user is authorized to
perform.

Happens(AuthorizeUser(SOME_USER), time) -> Happens(RevokeAuthorization(SOME_USER),
time+DELAY).
```

Further, the ***Auditing*** property requires that all the operations performed by the composition process should be logged and are available for auditing (if needed). In relation to the proposed event-calculus modeling approach, an event called *CreateLog()* can be used that should be invoked after each event to log the process state.

6.2.5 Dynamic Task Delegation

Task delegation presents one of the business process security leitmotifs. It defines a mechanism that bridges the gap between both workflow and access control systems. There are two important issues relating to delegation, namely allowing task delegation to complete, and having a secure delegation within a workflow. Delegation completion and authorization enforcement are specified under specific constraints. Constraints are defined from the delegation context implying the presence of a fixed set of delegation events to control the delegation execution.

We have proposed an approach that aims to reason about delegation events to specify delegation policies dynamically[Gaaloul 2010]. To that end, we present an event-based task delegation model to monitor the delegation process. We then identify relevant events for authorization enforcement to specify delegation policies. Moreover, we propose a technique that automates delegation policies using event-calculus to control the delegation execution and increase the compliance of all delegation changes in the global policy.

6.3 Example

We will now review the motivating example (Section-2.2) and discuss event-calculus modeling for different temporal and security requirements associated with the process. We identified different components for the motivating example in Section-4.5 and different conditions and control/data flow specification for the composition process in Section-5.7. In this section, we will further enrich the model by adding different temporal and security requirements.

For the temporal constraints, we first specify the exact delay between the invocation/start of two components. This is necessary as only specifying the dependency using the first dependency pattern, Section-5.1, would not enforce to invoke the dependent component immediately after the source component. However, this can be enforced using the second dependency pattern, Section-5.1, with *DELAY* having value 1. As an example we first review the control flow specification at the start of the process that we need to first check if the CRR/FC/SC are available and if CRR is available he can check if its just the false alarm. In the model below, the first axiom specifies the dependency to check the false alarm only if CRR is available. However it does not specify if indeed the CRR is available, when (after how much delay) the check for false alarm should be made. The last two axioms handle this behavior and are based on second dependency pattern as discussed in Section-5.1. The delay value in this case is 1 specifying immediately after, and we an have similar axioms for other dependent components which are omitted from the process specification.

Motivating example - Delay for checking the false alarm

Happens(EvalCondTrue(IsCRRAvailable), time) → Happens(EvalCondTrue(FalseAlarmCheckByCRR), time+1) | Happens(EvalCondFalse (FalseAlarmCheckByCRR), time+1).
Happens(EvalCondTrue(FalseAlarmCheckByCRR), time) | Happens(EvalCondFalse(FalseAlarmCheckByCRR), time) → Happens(EvalCondTrue (IsCRRAvailable), time-1).

The delay value can be further modified to specify delay between different components and conditions such as in the model below we handle the temporal requirement that CRR arrives 15 minutes after the *CallStaffWS* has been invoked.

Motivating example - If CRR not available, he arrives 15 minutes after CallStaffWS is invoked

Happens(Invoke_ synchservice(CallStaffWS), time) & !HoldsAt(ConditionTrue(IsCRRAvailable),time) → Happens(EvalCondTrue(HasCRRArrived), time+15) | Happens(EvalCondFalse(HasCRRArrived), time+15).
Happens(EvalCondTrue(HasCRRArrived), time) | Happens(EvalCondFalse(HasCRRArrived), time) → Happens(Invoke_ synchservice(CallStaffWS), time-15) & !HoldsAt(ConditionTrue(IsCRRAvailable),time).

We can have similar model as shown above for SC and FC, with their arrival time being planned to be 25 minutes after the invocation of *CallStaffWS*. Similarly, the fire-brigade arrives 15 minutes after the *FireBrigadeWS* is invoked and the condition that *HasFBArrived* is thus evaluated 15 minutes after the request to the *FireBrigadeWS* has been made.

The same pattern can be used to specify the response-time for a component such as once the fire-brigade arrives and they decide it is not a false alarm, the fire containment starts which may take 20 minutes (theses are only estimates and the actual containment time can be different depending upon for instance the scale of the fire). Th eevent-calclus model below specifies this temporal requirement:

ent finishes and once the CRR has arrived (or was already there), examine site activity starts. The last axiom in the model below handle this requirement and the examine site activity make take 10 minutes, we have omitted the axioms for specifying temporal constraint as they very similar to the second and third axioms in the model below (Start/End_activitywithstate abbreviated as Start/End_activityWS).

Motivating example - Fire-containment (with temporal constraint) and ExamineSite

Happens(Start_ activityWS(FireContainment), time) → Happens(End_ activityWS (FireContainment), time+20).
Happens(End_ activityWS(FireContainment), time) → Happens(Start_ activityWS (FireContainment), time-20).

Further, we consider that the *PlanRecovery* activity may take 15 minutes (again only just an estimate) and the activity for the recovery of the priority items (*RecoverPriorityItems*) can start once the planning is done and it may take 20 minutes to recover the items. In the model below, we also specify the composition goal that is to have the items recovered 90 minutes after the start of the process.

Motivating example - the composition goal

; composition goal
HoldsAt(Finished_ activitywithstate(RecoverPriorityItems), 90).

In terms of security requirements for the composition process we here consider a component (activity/service) based access control scheme. While modeling the control/data flow for the composition process we modeled the arrival/availability of emergency staff as conditions and defined axioms that enforce their availability for certain components to be started/invoked. As an example, we considered that the emergency services can either be invoked by CRR (with exception of some services which can be invoked automatically by the composition process). Then for checking a false-alarm either CRR should be there or fire-brigade has reached the site. For planning all the emergency staff (CRR/FC/SC) and others should be there. We will discuss how the security requirements can be updated and modified during process execution, later in Section-8.4.

6.4 Summary

In this chapter we have discuss how the proposed approach allows to model different security and temporal aspects related to the composition process. We have first discussed patterns for specifying temporal requirements, such as response-time, restart/refresh and ordering relations for the components based on the base-relations for the Allen's interval algebra. The table below provides an overview regarding the structure of this chapter, for event-calculus models for specifying different temporal requirements:

Temporal patterns	***Section for event-calculus model***
Response time	*Event-calculus based pattern for specifying the time taken by components to finish execution is discussed in Section-6.1.1.*
Restart/Refresh	*Pattern for specifying components re-invocation (either after some specific time or based on data invalidity) is presented in Section-6.1.2.*
Time frame	*EC Pattern for specifying the time frame allowed for a component's execution is presented in Section-6.1.3.*
Interval algebra	*Patterns based on base relations of Allen's interval algebra are discussed in Section-6.1.4.*
Time units	*An approach to cater different time-units used for specifying temporal aspects for different components is discussed in Section-6.1.5.*

Further, we have also presented different security requirements for the Web services composition problem and different Web services interaction levels at which they can be handled. We have then discussed how the proposed event-calculus based approach can be used for specifying different security requirements for the composition process, at different interaction levels. Then, we have discussed event-calculus models for specifying different temporal and security requirements for the motivating example, by using the patterns presented in this chapter. The generic pattern based approach has allowed to implement a Java-based application, called *ECWS*, that can automatically generate the event-calculus models for the process specification (including the temporal and security requirements and we will discuss the *ECWS* application in Section-9.2.

Part IV

VERIFICATION AND MONITORING

Chapter 7

Instantiation and Verification

Contents

The proposed components for the process specification, as discussed in Section-4.3, also include the *Nodes* that need to be discovered and instantiated to some concrete Web services based on some specified constraints. In this chapter we will first discuss the proposed SQWRL based nodes instantiation approach by also providing a brief background for the SQWRL. Then, as the proposed declarative event-calculus based composition process specification may only be partially defined (in terms of process fragments) and may contain conflicts or inconsistencies, we will also discuss the process instantiation which aims to find a solution for possibly partially defined composition process, connecting different process fragments respecting any functional and non-functional aspects associated with the process. Further, if there are some conflicts in the composition design and/or the specified constraints are too strict, this leads to empty solution set and we will then discuss the proposed approach for the verification of the composition process to identify any conflicts or hard constraints. Finally, we will discuss the proposed approach for composition process monitoring and recovery, while in execution.

7.1 Nodes instantiation

The nodes instantiation phase aims to find candidate services for the specified nodes in the composition process. Below we discuss a SQWRL based approach [Zahoor 2009b] to handle nodes instantiation which uses SQWRL queries to search through the OWL-S based repository for Web services that satisfy the constraints associated with the nodes. The nodes are thus resolved using the SQWRL based approach and are added to the event-calculus based process description. Before going into the details of the proposed approach, we briefly discuss some background about the OWL-S and the SQWRL.

7.1.1 Background

OWL-S is a OWL-based Web service ontology providing constructs for describing a Web service in terms of a service profile (which describes what a service provides and allows service classification using *ServiceCategory* attribute and specification of non-functional properties using *ServiceParameter* attribute), the process model (which describes how the service works) and the service grounding (which specifies concrete details such as message formats, network addresses used and others). The Semantic Web Rule Language (**SWRL**) is intended to be the rule language for the semantic web. The SWRL rules are written in terms of OWL classes, properties and individuals. SWRL also provides a set of core built-ins for strings manipulation, basic mathematical operations and others. It also allows to extend the core built-ins to add user defined built-ins. An example of a SWRL rule to express that a person with a older female sibling has a older sister can be written as:

SWRL example)

Person(?p) ∧ hasAge(?p, ?pAge) ∧ hasSibling(?p,?s) ∧ Woman(?s)
∧ hasAge(?s, ?sAge) ∧ swrlb:greaterThan(?sAge, ?pAge) → hasOlderSister(?p,?s)

In the rule above, *Person* is a class with a sub-class named *Woman* and *hasSibling* and *hasOlderSister* are OWL properties with domain and range of the class *Person*. The rule also uses the *hasAge* property (with domain as *Person* and range of primitive datatype *Integer*) and the SWRL builtin (*swrlb:greaterThan*) to add *hasOlderSister* property to all individuals who have older female siblings. The Semantic Query-Enhanced Web Rule Language (**SQWRL)** adds querying capabilities to SWRL by providing primitives to select, count and perform other operations on the results of a SWRL rule. Finally, SQWRL queries (and so as SWRL rules) require a rule-solver and for that we have used the JESS rule-solver.

7.1.2 The proposed approach

The proposed SQWRL based approach requires the services descriptions of the candidate services to be specified using OWL-S, which include the specification of the non-functional properties using the approach specified in DAML[1] examples. However, our proposal can also adapt the various QoS extensions to the OWL-S such as QoS-MO [Tondello 2008], QoSOnt [Dobson 2005] and other approaches that extend OWL-S for specifying QoS properties. Then, the SQWRL queries are used to search through the OWL-S service model to get the services which match the associated local constraints as shown in the example below.

Nodes instantiation using SQWRL

service:Service(?someServiceName) ∧ service:presents(?someServiceName, ?someServiceProfile) ∧ profile:serviceCategory(?someServiceProfile, ?someServiceCategory) ∧ profile:code(?someServiceCategory, ?categoryCode) ∧ swrlb:equal(?categoryCode,) ∧
profile:serviceParameter(?someServiceProfile,?SomeServiceParameter)∧
profile:sParameter(?SomeServiceParameter, SomeParameterValue) ∧ ...
other constraints
→sqwrl:select(?someServiceName)

The query first selects the candidate services for a particular node based on the Web service classification code specified in the *serviceCategory* attribute of the service *profile*. We have used NAICS[2] categorization for the Web services. So the initial part of the query will select only the services that have the same type as specified in the local constraints for the node. Next, we filter for the non-functional constraints, specified using the *ServiceParameter*, such as reliability.

The node instantiation result may be a collection of Web service and in case of a loosely constrained node, the result set can be very large. Our proposal thus aims to choose the best matched Web service based on some user-specified criteria such as the quality rating for the Web services, by assuming that some trusted third-party has quality ratings assigned to services. This choice can also be based on some other non-functional requirements or the user can also manually select the best matched (or suited) Web service. Once the candidate services and the best matched (or suited) Web service have been identified using the SQWRL based approach they are added to event calculus as other service instances with one exception, axioms are added to event calculus model for invoking only the best matched service from the candidate services and for not invoking the other candidates. The event-calculus pattern below can be used to handle this behavior:

Pattern for specifying candidate services after nodes instantiation)

;for all candidate services other than selected Web service
Happens(Invoke_synchservice (ALL_CANDIDATES),time).

An important aspect regarding nodes is how to deal with dependencies amongst nodes with a node being dependent on some other node and thus requiring to cater

[1]http://www.daml.org/services/owl-s/1.1/examples.html
[2]http://www.census.gov/eos/www/naics/

for this dependency while instantiating nodes. In this case, if the instantiation result set for a node is empty and if the *worksWith* relation is unsatisfied, we need to backtrack to the results of dependent node to select some other instantiation solution and then proceed to finding solution for the current node. The process continues until all backtrack solutions have been explored.

7.1.3 The *worksWith* dependency

We consider that a service *worksWith* some other service using the modified form of the composability rules discussed in [Narayanan 2002a]. These rules consider the syntactic and semantic properties of Web services. Syntactic rules include the rules for operation modes (one-way, request-response. . .), and the rules for binding protocols and data formats of interacting services. The Semantic rules include the *message composability* rule which defines that two Web services are composable only if the output message of one service is compatible with the input message of another service. In case of semantic Web services described using OWL-S, it is important to consider that the input and output parameters are defined in the domain ontology as specifying them as datatypes add very little to semantics ([Redavid 2008] has a detailed discussion). Further, the *operation semantic composability* defines the compatibility between the domains, categories and purposes of two services while the *qualitative composability* defines the requester's preferences regarding the quality of operations for the composite service. Then, the *composition soundness* considers whether a composition of services is reasonable, see [Narayanan 2002a] for details.

7.1.4 Backtracking and propagation

The backtracking process involves finding an alternative to some previously chosen node instantiation solution. Backtracking is needed when the *worksWith* relation for some node is unsatisfied resulting in empty result set. Once the backtracking process execution terminates, resulting in a newly chosen solution (instance), the composition solution must be recomputed and may require the propagation of newly chosen solution. This would likely be the case when a (partial) solution to the composition process has already been determined and backtracking to some higher node (in hierarchal order) may result in propagating the new solution. Further, propagation may also be needed when the user fine tunes the solution by manually selecting some other Web service after the instantiation process. The propagation process will require recomputing the composition and this may result in significant overhead to re-instantiate the service nodes. Our proposal aims to re-instantiate only the Web service nodes that have dependency on the node to backtrack (either a *worksWith* dependency or a data dependency).

7.2 Process instantiation

Once all the nodes added to the composition process have been resolved and selected services are added to the event-calculus based composition design, an event-calculus 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. As similar to the nodes instantiation, 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 and others. In reference to temporal properties, one criteria for solution-selection is minimal time requiring to find a solution specifying to complete the execution process in minimal possible time. Below we highlight guidelines that help enforcing this criteria:

- For all the services having no dependency, they should be invoked at the start of the process (concurrently if possible).
- Asynchronous services should be invoked (if possible) before the synchronous ones and invoking them at the start will help invoking the synchronous ones while the response is yet to be received from asynchronous ones.
- For the services having dependency on some other services, they should be invoked as soon as the dependency is resolved.

7.2.1 Example

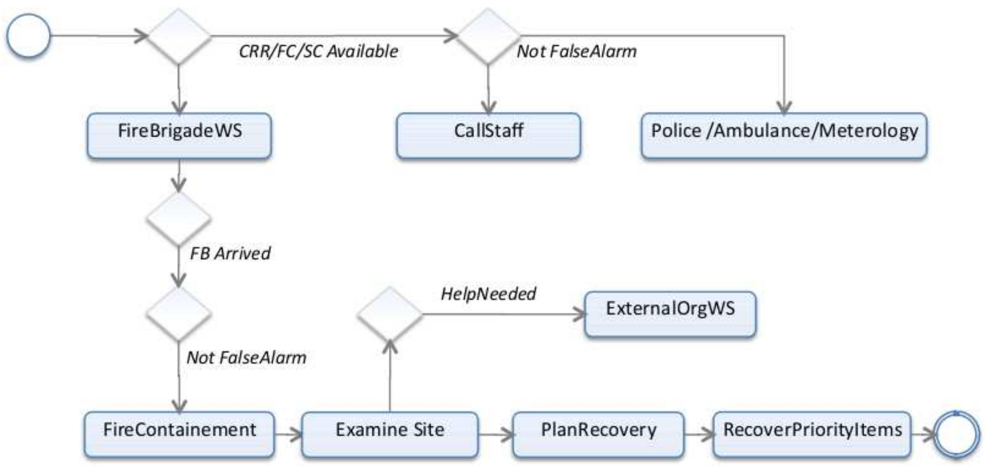


Figure 7.1: Instantiated model for the motivating example

Let us now review the motivating example and invoking the *DECReasoner* for the event-calculus model for the motivating example presented in Section-2.2, gives us a set of solutions including the one shown below (*Condition* abbreviated as *Cond*, *activity(with/without)state* abbreviated as *activity(S/WS)* and *Invoke_ synchservice* abbreviated as *Invoke*) :

Instantiation (solution finding) for the motivating example

```
0
Happens(EvalCondTrue(IsCRRAvailable/IsFCAvailable/IsSCAvailable), 0).
Happens(Start_ activityWS(Start), 0).
1
+CondTrue(IsCRRAvailable/IsFCAvailable/IsFCAvailable). +Finished_ activityWS(Start).
Happens(EvalCondFalse(FalseAlarmCheckByCRR), 1).
2
+CondFalse(FalseAlarmCheckByCRR).
Happens(Invoke(AmbulanceWS/CallStaffWS/FireBrigadeWS/MeteorologyDeptWS/PoliceWS), 2).
3
+RespReceived_ serv(AmbulanceWS/CallStaffWS/FireBrigadeWS/MeteorologyDeptWS/PoliceWS).
4
...
17
Happens(EvalCondTrue(HasFBArrived), 17).
18
+CondTrue(HasFBArrived). Happens(EvalCondFalse(FalseAlarmCheckByFB), 18).
19
+CondFalse(FalseAlarmCheckByFB). Happens(Start_ activityS(FireContainment), 19).
20
+Started_ activityS(FireContainment).
21
...
39
Happens(End_ activityS(FireContainment), 39).
40
-Started_ activityS(FireContainment). +Finished_ activityS(FireContainment).
Happens(Start_ activityS(ExamineSite), 40).
41
+Started_ activityS(ExamineSite).
42
...
45
Happens(End_ activityS(ExamineSite), 45).
46
-Started_ activityS(ExamineSite). +Finished_ activityS(ExamineSite).
Happens(EvalCondTrue(IsExternalHelpNeeded), 46).
47
+CondTrue(IsExternalHelpNeeded). Happens(Invoke(ExternalOrgWS), 47).
48
+ResponseReceived_ serv(ExternalOrgWS). Happens(Start_ activityS(PlanRecovery), 48).
49
+Started_ activityS(PlanRecovery).
50
...
63
Happens(End_ activityS(PlanRecovery), 63).
64
-Started_ activityS(PlanRecovery). +Finished_ activityS(PlanRecovery).
Happens(Start_ activityS(RecoverPriorityItems), 64).
```

```
65
+Started_ activityS(RecoverPriorityItems).
66
...
84
Happens(End_ activityS(RecoverPriorityItems), 84).
85
-Started_ activityS(RecoverPriorityItems). +Finished_ activityS(RecoverPriorityItems).
```

7.3 Process verification

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 to verify the composition process. In this section, we will first discuss the motivation for the proposed SAT based symbolic model checking approach and then will discuss some of the properties that can be verified using the proposed approach. Further, as the conflict clauses returned by the SAT solver can be very large, we will also discuss the proposed filtering criteria to reduce the clauses to identify the nature of conflicts.

7.3.1 Motivation

Traditional approaches for the Web services composition rely on a workflow-based approach where the process is modeled using approaches such as BPMN and are later translated to approaches such as WS-BPEL for their execution. The traditional approaches for modeling the composition process are very intuitive and make it easier to model the processes however this ease is coupled with lack of strictness and the process specification includes arbitrariness allowing different interpretations for a single process. As a result a number of approaches have been approaches to define strict semantics to the processes in order to formally verify them.

Further, the traditional workflow-based approaches for the Web services composition are highly procedural as they over-constrain the composition process making it rigid and difficult to handle dynamically changing situation . In contrast some declarative approaches have been proposed in the literature [Zahoor 2010a, van der Aalst 2006] that require the specification of constraints that mark the boundary of any solution to the composition process and any solution that respects the constraints associated with the composition process is considered a valid solution. In addition, the traditional approaches in general require mapping the process (defined using procedural approaches such as WS-BPEL) to some formal logic and then verifying the process and this makes it difficult to verify the non-functional properties (such as temporal and security requirements) associated with the composition process as for first, it is difficult to specify the non-functional properties using the traditional approaches such as BPEL and a number of approaches have been proposed as an extension to WS-BPEL for specifying non-functional aspects and then to add formal semantics to them for their verification.

Specifying the exact and complete sequence of activities to be performed for the composition process, as required by the traditional procedural approaches, however does make it possible to use proposed automata or petri-nets based approaches for design-time verification of composition process. However with the declaratives approaches the process may be only partially defined and thus this makes it difficult to use traditional approaches for the process verification as the transition system for a declarative process can be very large as all the transitions have not been explicitly defined (as by traditional procedural approaches). This motivates the use of symbolic model checking using satisfiability solving for process verification, instead of using explicit representation of state transition graphs and/or using the binary decision diagrams. The verification properties can include the connectivity, compatibility and behavioral correctness (safety and liveness properties) and the proposed approach allows for both model checking the verification properties and for identifying and resolving the conflicts in the process specification as a result of process.

7.3.2 Verification properties

The proliferation of proposed approaches for composition process verification has lead to numerous verification properties that are dependent on some specific case-study [Röglinger 2009] such as the fitness property [Rozinat 2008] requires to identify if the behavior of a composition process conforms to its implementation, the usability verification property as proposed by [Schlingloff 2005, Martens 2005] requires the composition process to terminate properly, syntactic compatibility [Martens 2005] requires to identify if Web services can be composed (and thus are compatible) with respect to their interfaces, while the semantic compatibility requires to identify if the composition of Web services satisfies usability.

Some seminal work on categorizing the verification properties and defining correctness of the composition process can be found in [Röglinger 2009].The authors have categorized the verification properties in application dependent and application independent categories and have also classified them as structural or behavioral correctness categories. Structural correctness requires that for each operation by which a composition process is defined and exposes its functionality to other Web services interface there is at least one identically named operation in participating component Web services interface that matches with respect to number, sequence, and types of parameters. Structural correctness can be verified by comparing the WSDL-based interfaces of participating Web services and the invocation statements WS-BPEL process specification or if the Web services are semantically annotated, operations also need to be matched with respect to input and output parameters.

Structural correctness serves as the pre-requisite for verifying the behavioral correctness of the composition process which requires that the sequences of messages to conform to the specified safety and liveness properties [Ouaknine 2005]. Safety claims must not be violated while the liveness claims must always hold [Holzmann 2004]. Further, in terms of composition process verification there is need to classify the behavioral claims into application-independent (generic issues like mutual exclusion and deadlock freedom) and application-dependent (based on specific use-cases) categories [Röglinger 2009].

7.3.3 The proposed approach

For the proposed approach, the verification properties are added to the process specification (in terms of event-calculus axioms) and then the reasoner is invoked for instantiating the process, which indeed is the connectivity verification property. As the composition process is (possibly) partially defined the basic verification objective is to verify that if a solution exists that respects all the associated functional and non-functional constraints for the process. In other words, we must ensure that the fragments of (possibly) partially defined process can be connected in a way that the orchestration is compatible with the constraints associated with the composition process.

Connectivity and computability can ensure that there exists a solution to composition process and is indeed a necessary condition for defining the correctness of the process. A connected process may represent a solution but as the objective of the proposed approach is to handle dynamicity, process change and the ability to self-heal and adapt to continuously changing situation by finding alternatives based on current process state, the connectivity itself cannot be considered as a sufficient condition. We can thus augment the verification properties to also include the behavioral properties, such as the liveness and safety properties. Connectivity and computability can ensure the safety properties, for the initial solutions returned, as the process is connected and solution is returned only if safety claims hold, while the liveness properties can ensure that the alternatives to the solutions returned (if needed to cater for process change during execution) can still hold the safety claims or not or which claims are violated.

The violations to verification properties, hard constraints or incorrect process specification leads to conflicts that need to be identified and resolved. The conflicts in process specifications can be broadly categorized into the syntactic and semantic categories. The syntactic conflicts result due to erroneous process specification and not following the syntactic rules for process specification using Discrete Event-Calculus Language, such as not following the naming conventions for instances, events or fluents definition. The DCReasoner allows to identify the syntactic errors providing error description that can be used to rectify the syntactic errors.

The semantic conflicts result from the process specification including deadlocks, hard and conflicting constraints. The specified composition goal (to hold at specified time-point) may not be possible to achieve if the dependency between two components cannot be respected, within the specified time-frame. For instance, it may be the case that starting two activities in parallel can achieve some specified goal however executing them in sequence can lead to failure. The conflicts can be based on local temporal constraints, such as the response-time for different components can be higher to achieve the specified goal at specified time-point. Further the conflicts can appear also due to the temporal constraints specified between different components, by the composition process. Further, the conflicts can be based on security constraints associated with the composition process such as the SoD constraint requiring prohibition to invoke of a service if another service had been executed, possibly combined with temporal conditions (e.g. the ban lasts only two hours), access control aspects such as the permission/prohibition to invoke a service given a role.

7.3.4 Filtering the unsatisfiable-core

The event-calculus 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 un-satisfiable clauses (termed as unsatisfiable core) can be very large and we thus propose to filter the unsatisfiable core, but before going into the details of the filtering approach we first briefly discuss the event-calculus to SAT encoding process.

The event-calculus to SAT encoding process is detailed in [Mueller 2006] and it works by first applying syntactic transformations to all the input formulas containing the predicate symbols such as Initiates, Terminates, Releases, or Trajectory to reduce resulting SAT problem size and the transformed formulas are added to the conjunction of problem formulas. Then, conjunction of any problem formulas not modified by syntactic transformations are also added and the Happens predicates are completed in the conjunction of problem formulas. Further, in order to enforce the commonsense law of inertia, explanation closure frame axioms from Initiates, Terminates, and Releases axioms are added to the conjunction of problem formulas. Then, the conjunction of problem formulas are transformed into a propositional calculus formula and finally ground atoms are mapped to the variables of the satisfiability problem. A detailed discussion with corresponding example can be found in [Mueller 2006].

In order to filter the unsatisfiable core, we propose to only consider the encoded clauses added for formula axioms (that specify the control/data flow, temporal, security or other constraints for the particular problem). This allows us to ignore the encoded clauses added for the frame and completion axioms, encoding of Initiates/Terminates symbols, initial conditions for fluents, the composition goal and the initial conditions for Releases axioms. The use of pattern based approach, further allows to ignore the encoding of generic axioms needed to model the services and activities.

7.3.5 Example

The instantiated model for the composition design as presented in Section-7.2.1 shows that the composition problem does have a solution and it satisfies the connectivity and safety properties associated with the process. However, the process can also be augmented with liveness properties to explore other paths and cases. As an example we consider the case of adding the liveness property that the emergency staff is not there and reaches at the site once the CallStaffWS has been invoked, as shown below:

Motivating example verification - adding liveness properties
{time} Happens(EvalConditionFalse(Is(SC/FC/CRR)Available), time).

The event-calculus axiom shown requires the reasoner to consider that the condition to check the availability of the emergency staff should not hold at any time-point (notice the existential quantification of the time variable). It is also possible to modify and further constrain other aspects such as modifying the time taken by fire-containment and others to conduct a design-time analysis for the composition process. The result returned by the reasoner with the addition of the liveness property is shown below (*Condition* abbreviated as *Cond*, *activity(with/without)state* abbreviated as *activity(S/WS)* and *Invoke_synchservice* abbreviated as *Invoke*):

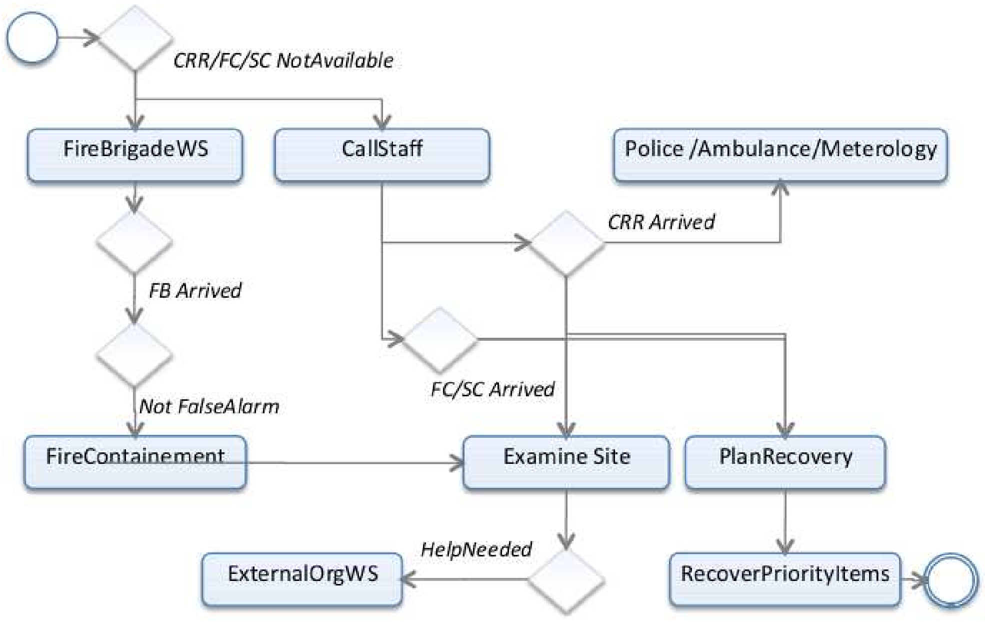


Figure 7.2: Liveness properties verification for the motivating example

Liveness properties verification: instantiated model

```
0
Happens(Start_activityWS(Start), 0).
Happens(EvalCondFalse(IsCRRAvailable/IsFCAvailable/IsSCAvailable), 0).
1
+CondFalse(IsCRRAvailable/IsFCAvailable/IsFCAvailable). +Finished_activityWS(Start).
Happens(Invoke(CallStaffWS), 1). Happens(Invoke(FireBrigadeWS), 1).
2
+RespRecvd(CallStaffWS).
+RespRecvd(FireBrigadeWS).
...
16
Happens(EvalCondTrue(HasCRRArrived), 16). Happens(EvalCondTrue(HasFBArrived), 16).
17
+CondTrue(Has(CRR/FB)Arrived). Happens(EvalCondFalse(FalseAlarmCheckByFB), 17).
Happens(Invoke(AmbulanceWS/MeteorologyDeptWS/PoliceWS), 17).
18
+CondFalse(FalseAlarmCheckByFB).
+RespRecvd(AmbulanceWS/MeteorologyDeptWS/PoliceWS).
Happens(Start_activityS(FireContainment), 18).
19
+Started_activityS(FireContainment).
...
26
Happens(EvalCondTrue(HasFCArrived/HasSCArrived), 26).
27
+CondTrue(HasFCArrived). +CondTrue(HasSCArrived).
...
38
Happens(End_activityS(FireContainment), 38).
39
-Started_activityS(FireContainment). +Finished_activityS(FireContainment).
Happens(Start_activityS(ExamineSite), 39).
40 +Started_activityS(ExamineSite).
...
44
Happens(End_activityS(ExamineSite), 44).
45
-Started_activityS(ExamineSite). +Finished_activityS(ExamineSite).
Happens(EvalCondTrue(IsExternalHelpNeeded), 45).
46
+CondTrue(IsExternalHelpNeeded). Happens(Invoke(ExternalOrgWS), 46).
47
+RespRecvd(ExternalOrgWS). Happens(Start_activityS(PlanRecovery), 47).
48
+Started_activityS(PlanRecovery). ...
62
Happens(End_activityS(PlanRecovery), 62).
63
-Started_activityS(PlanRecovery). +Finished_activityS(PlanRecovery).
Happens(Start_activityS(RecoverPriorityItems), 63).
64
+Started_activityS(RecoverPriorityItems).
...
83
Happens(End_activityS(RecoverPriorityItems), 83).
84
-Started_activityS(RecoverPriorityItems). +Finished_activityS(RecoverPriorityItems).
```

Further, in order to demonstrate the process verification in case of a conflict in process specification and corresponding filtering of the unsatisfiable-core, we first consider a simple example with two synchronous services, AmbulanceWS and PoliceWS that need to be invoked (in parallel at time-point 0) to have response from at time-point 1. However, we intentionally add a conflicting axiom to have the activity PoliceWS dependent on service AmbulanceWS. The event-calculus model for the composition process is shown below (*Invoke_synchservice* abbreviated from actual model as *Invoke* and *ResponseReceived_synchservice* abbreviated as *RespRecvd*):

Process specification with a conflict

load includes/synchservice.e
synchservice AmbulanceWS, PoliceWS

;the conflicting axiom
Happens(Invoke(PoliceWS), time) → HoldsAt(RespRecvd(AmbulanceWS), time).
HoldsAt(RespRecvd(synchservice), 1).

Invoking the SAT-solver for the process verification gives us a set of unsatisfiable clauses, as shown below. The unsatisfiable core returned by the reasoner contains 7 clauses for a very simple composition process and with the increase in problem complexity and size, the unsatisfiable core can be very large. However, the proposed filtering on the unsatisfiable core can be very effective for reducing the size.

Unsatisfiable-core for the example

7 unsatisfied clauses:
-10 0: (!ReleasedAt(RespRecvd(PoliceWS), 0)).
-8 0: (!HoldsAt(RespRecvd(PoliceWS), 0)).
-7 0: (!HoldsAt(RespRecvd(AmbulanceWS), 0)).

4 6 8 -5 0: (Happens(Invoke(PoliceWS), 0) | ReleasedAt(RespRecvd(PoliceWS), 1) | HoldsAt(RespRecvd(PoliceWS), 0) | !HoldsAt(RespRecvd(PoliceWS), 1)). 5 0: HoldsAt(RespRecvd(PoliceWS), 1).

7 -4 0: (HoldsAt(RespRecvd(AmbulanceWS), 0) | !Happens(Invoke(PoliceWS), 0)).
10 -6 0: (ReleasedAt(RespRecvd(PoliceWS), 0) | !ReleasedAt(RespRecvd(PoliceWS), 1)).

For the above mentioned unsatisfiable core, the clauses *4 6 8 -5 0* and *10 -6 0* are the frame axioms so they can be ignored. Then the axioms *-10 0*, *-8 0* and *-7 0* are the initial conditions for the fluents and initial conditions for the ReleasedAt. Further, the axiom *5 0* is the composition goal and the only clause left is *7 -4 0: (HoldsAt(RespRecvd(AmbulanceWS), 0) | !Happens(Invoke(PoliceWS), 0)).* This clause is transformed form of the dependency axiom in the process specification requiring PoliceWS to started if the AmbulanceWS has finished and this clause is causing the conflict because respecting the dependency, it is not possible to achieve the composition goal. However, removing the dependency will allow services to be invoked in parallel and thus achieve the composition goal.

The unsatisfiable core for the simple example contained only 7 conflict clauses and they were filtered to have only one clause. The conflict clause is in fact based on the proposed patterns for specifying the dependency amongst components, as discussed in Section-5.1 and the use of pattern based approach thus further allows us to concentrate only on specific kind of clauses, once the unsatisfiable core has been filtered. Let us now move to the motivating example and introduce a cyclic dependency between the conditions to check CRR availability and the condition to check if there is a false alarm, as shown in the model below:

Motivating example with deadlock

```
;check if it is a false alarm only if CRR is available
Happens(EvalCondTrue(FalseAlarmCheckByCRR), time) | Happens(EvalCondFalse (FalseAlarm-
CheckByCRR), time) → HoldsAt(CondTrue(IsCRRAvailable),time).

;the conflicting axiom - deadlock
Happens(EvalCondTrue(IsCRRAvailable), time) | Happens(EvalCondFalse(IsCRRAvailable), time)
→ HoldsAt(CondTrue(FalseAlarmCheckByCRR),time).
```

Invoking the SAT-solver for the process verification gives us a set of unsatisfiable clauses as shown below:

Unsatisfiable-core for the motivating example with deadlock

```
5 unsatisfied clauses: -11125 0: (!HoldsAt(CondTrue(FalseAlarmCheckByCRR), 0)).
4215 0: Happens(Start_ activityWS(Start), 0).
5118 8118 -4215 0: (Happens(EvalCondTrue(IsCRRAvailable), 0) | Happens(EvalCondFalse (Is-
CRRAvailable), 0) | !Happens(Start_ activityWS(Start), 0)).

11125 -5118 0: (HoldsAt(CondTrue(FalseAlarmCheckByCRR), 0) | !Happens (Eval-
CondTrue(IsCRRAvailable), 0)).
11125 -8118 0: (HoldsAt(CondTrue(FalseAlarmCheckByCRR), 0) | !Hap-
pens(EvalCondFalse (IsCRRAvailable), 0)).
```

Analyzing the unsatisfiable core for the above example and focusing only on the clauses for dependency pattern (the last two clauses), we can identify that they are the transformed clauses for the newly added axiom causing the deadlock, *Happens(EvalCondTrue(IsCRRAvailable), time) | Happens(EvalCondFalse (IsCRRAvailable), time) → HoldsAt(CondTrue(FalseAlarmCheckByCRR),time).* The unsatisfiable core, however is not always very small and may contain a number of clauses. As an example now we add a conflicting goal to the process specification as shown below:

Motivating example specification with conflicting goals

```
HoldsAt(Finished_ activitywithstate(RecoverPriorityItems), 90).
HoldsAt(Finished_ activitywithoutstate(End2), 90).
```

Invoking the SAT-solver for the process verification gives us a large set of 2002 unsatisfiable clauses primarily because of grounding of the clauses to all the time-points. However, by filtering, ignoring multiple groundings for the same conflict clause and only focusing on the conflict clauses for specifying dependency pattern, we narrow the unsatisfiable core to following clauses:

Filtered unsatisfiable-core for the motivating example with conflicting goal

```
2 -6921 0: HoldsAt(RespRecvd(FireBrigadeWS),1) | !Happens(EvalCondTrue(HasFBArrived),1).
1808 -3019 0: HoldsAt(Started_actS(FireContainment),1)|!Happens(End_actS(FireContainment),1)
1817 -3028 0: HoldsAt(Started_actS(RecoverPriorityItems),1)|!Happens(End_actS (RecoverPriorityItems), 1).
1814 -3025 0: HoldsAt(Started_actS(PlanRecovery),1) | !Happens(End_actS (PlanRecovery), 1)
4216 -4215 0: HoldsAt(Finished_actS(Start),1) | !Happens(Start_actWS(Start),0).

11118 11121 -1822 0: HoldsAt(CondTrue(IsCRRAvailable), 0) | HoldsAt(CondTrue (HasCRRArrived), 0) | !Happens(Start_activityS(ExamineSite), 1).
11120 11122 -1813 0: HoldsAt(CondTrue(IsSCAvailable), 0) | HoldsAt(CondTrue(HasSCArrived), 0) | !Happens(Start_activityS(PlanRecovery), 0).
11128 11135 -1 0: HoldsAt(CondFalse(IsCRRAvailable), 0) | HoldsAt(CondFalse( FalseAlarmCheckByCRR), 0) | !Happens(Invoke(FireBrigadeWS), 0).

7518 -4230 0: Happens(EvalCondTrue(FalseAlarmCheckByFB),0) | !Happens (Start_activityWS(End2),1).
11124 -10518 0: HoldsAt(CondTrue(HasFBArrived),0) | !Happens(EvalCondFalse (FalseAlarmCheckByFB),0).
4212 -7818 0: HoldsAt(Finished_activityS(ExamineSite),0) | !Happens(EvalCondTrue (IsExternalHelpNeeded),0).

11126 -4221 0: HoldsAt(CondTrue(FalseAlarmCheckByFB), 0) | !Happens(Start_activityWS(End2), 0).
11136 -1807 0: HoldsAt(CondFalse(FalseAlarmCheckByFB), 0) | !Happens(Start_activityS(FireContainment), 0).
```

From the filtered unsatisfiable core mentioned above, the last two axioms specify that in order to have the conflicting goal achieved, the condition *FalseAlarmCheckByFB* should be both evaluated to true and false and this highlights the conflict in specification.

7.4 Summary

The proposed components for the process specification, as discussed in Section-4.3, also include the *Nodes* that need to be discovered and instantiated to some concrete Web services based on some specified constraints. In this chapter, we have first discussed a SQWRL based nodes instantiation approach by also providing a brief background for the SQWRL. The choice of using SQWRL based approach is based upon the observation that the event-calculus based approach for nodes instantiation may incur some over-head, as we discussed in Section-4.3, and an external approach, where nodes are resolved and candidates Web services are added to the process specification before instantiating the process seems more efficient.

Then, as the proposed declarative event-calculus based composition process specification may only be partially defined (in terms of process fragments) and may contain conflicts or inconsistencies, we have discussed the process instantiation which aims to find a solution for possibly partially defined composition process, connecting different process fragments respecting any functional and non-functional aspects associated with the process. Further, if there are some conflicts in the composition design and/or the specified constraints are too strict, this leads to empty solution set and we have discussed the proposed SAT approach for the verification of the composition process to identify any conflicts or hard constraints. Further, the set of conflict clauses returned by the SAT solver (called unsatisfiable-core) can be very large and we have discussed the filtering criteria based on the patterns and the structure of conflict clauses. Further, we have also reviewed the motivating example and discussed process instantiation and verification for the motivating example.

CHAPTER 8

Monitoring and recovery

Contents

The instantiated solution(s) returned by the reasoner serves as a plan for the process execution by mapping different events and axioms defined for specifying the composition process to actual actions to be taken by process run-time. For instance, the defined events to invoke services can be mapped to actual service invocation calls by the process run-time. Further, once the process is in execution the need to monitor the Web services composition process during execution stems from two major objectives. At one hand continuously monitoring the resource utilization, SLA's violation, or some domain specific Key Performance Indicators (KPI's) may be required to measure the performance or to fulfill some domain specific monitoring requirements. Then, as the Web services are autonomous and only expose their interfaces, composition process is based on design level service contracts and the actual execution of composition process may result in the violation of the design-level services contracts due to errors such as network or service failures, change in implementation or other unforeseen situation. This highlights the need to detect the errors and react accordingly to cater for them. The reaction may include to calculate the effect the violation has on the overall process execution and then to recover from it.

The proposed event-based monitoring framework [Zahoor 2011] allows to specify and reason about the monitoring properties during composition process execution. The composition process is specified using the event-calculus and is then used to instantiate, verify and execute the composition process (see Figure 8.1-①). The instantiation phase involves finding a solution to the composition process using the event calculus reasoner and the instantiated plan is then executed using the execution engine (see Figure 8.1-②).

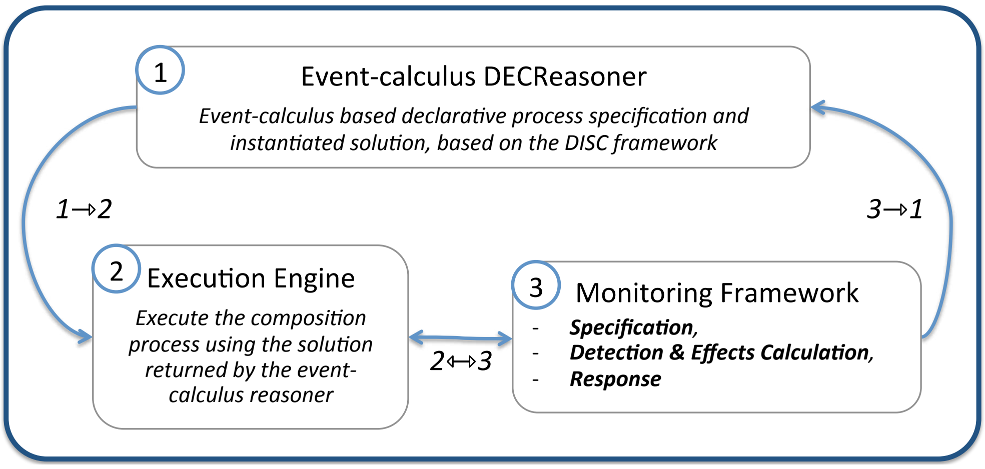

1
Event-calculus DECReasoner
Event-calculus based declarative process specification and instantiated solution, based on the DISC framework
1→2
3→1
2
Execution Engine
Execute the composition process using the solution returned by the event-calculus reasoner
2↔3
3
Monitoring Framework
- Specification,
- Detection & Effects Calculation,
- Response

Figure 8.1: Proposed monitoring framework

The proposed monitoring framework (see Figure 8.1-③) works during the composition process execution and is divided into three phases. The *specification* phase requires the user to specify the functional and non-functional properties that needs to be monitored to identify anomalies or needed for KPI's measurement. Then, the detection and effects calculation phase is both responsible for detecting any violations based on the specified properties and to calculate the side-effects the detected violation has on the overall process. Then, the *response* phase uses the user-specified actions to respond to the monitored property. In the sections to follow, we will first discuss the monitoring properties specification in Section-8.1 and then will discuss how the detection and effects calculation works once a violation is detected, in Section-8.2. Then, we will discuss the possible response actions to cater for the monitoring properties, in Section-8.3.

8.1 Properties specification

The *specification* phase requires the user to specify the functional and non-functional properties that needs to be monitored to identify anomalies or needs for KPI's measurement. The properties that need to be monitored are added to process description either at the process design (if they are already known, Figure-8.1-①) or they can be added to the process specification at the execution time. In the later case the process specification is updated and an updated instantiated solution is sought, in order to verify any conflicts and to get an updated execution plan as a result of process change during execution (see Figure 8.1–*3→1*).

Properties that can be monitored include the functional aspects such as monitoring the invocation and execution order or they can be based on non functional aspects such as temporal aspects requiring to monitor the response time for a service, delay between successive invocations of the service or monitoring invocation time-frame for a service. Further, the properties can also be based on data such as monitoring the data availability, validity and expiry or based on the security properties such as monitoring the data integrity, confidentiality, access-control. The choice of highly expressive event-calculus formalism even allows to combine the properties related to temporal, security and other aspects such as monitoring the data validity and access control within specific time frame which may be needed for instance, during dynamic task delegation (see [Gaaloul 2010] for details).

8.2 Detection and effects calculation

8.2.1 Detection

The detection of the violations can be handled at different levels using the proposed framework. At a basic level we first consider the violations to the execution plan, which is handled by maintaining an *event repository* which keeps track of all the messages exchanged between the composition process and the participating services during process execution. This repository is then used to find any mismatch between the temporal ordering of actual events and the ones mentioned in the initial instantiated plan. Using the basic detection technique, it is possible to find violations to the execution plan or the invocation and execution order of the services. However such a detection level may not be useful in detecting data values based or other low-level violations, as using the event-calculus, the process is modeled at an abstract level. This can be handled by also abstracting the processing of verifying the data values and other low level service details by using event-calculus fluents. For instance, we can have a fluent *ResponseValid(SomeService)* and an event called *ValidateResponse(SomeService)*, and whenever data is received from a service we check for its validity. Then, if the data is not considered valid, based on application level checks on data, the fluent *ResponseValid(SomeService)* does not hold and in-turn results in a mismatch between the initial instantiated plan and actual service execution. The detection phase may thus require the execution engine support (for instance checking data validity, see Figure 8.1-②).

Then, in order to detect the monitoring properties added at the execution time (e.g. based on external events not there in the initial instantiated plan), the "abduction reasoning" mode can be used by adding the newly added events and monitoring properties to the process model and re-invoking the reasoner. In case of no conflict and violation, the reasoner returns an updated plan based on the added events and monitoring axioms. However, if there is some conflict based on addition of new events or if the newly added monitoring property is not satisfied, the reasoner returns a set of unsatisfied clauses highlighting the error. The detection phase may thus also require the reasoner support (see Figure 8.1–*3*→*1*).

8.2.2 Effects calculation

Once a violation to some monitoring property is detected, the effects calculation phase is responsible for calculating the side-effects this violation has on the overall process flow. This allows to prioritize the violations based on their impact and it may be possible to ignore some violations, for instance if the response time delay for a service has no effect on the overall process goal and other functional and non-functional properties associated with the composition process. As the proposed approach allows to reason about the composition process and as the approach is based on event-calculus with different reasoning modes, the effects calculation is achieved by adding the partial plan with the violation to the initial plan and re-invoking the reasoner. Although the process may seem similar to the detection of monitoring properties added at the execution-time, there is one major difference; instead of using the "abduction reasoning" we use "deduction reasoning" in the effects calculation phase. This may further allow to foresee any anomalies which may not be evident now but may happen later in the process execution. The effects calculation phase thus requires the support from the event-calculus reasoner to perform deductive reasoning (see Figure 8.1–*3*➝*1*).

8.3 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) needed for the successful process execution. Then, in order to cater for the monitoring violations detected at the execution time, different recovery actions can be used in-order to recover from the violation. These actions may include to ignore the violation, terminate the process, re-invoke or substitute the service, find an alternative solution based on current process state or backtrack to some previous state and then seek an alternative solution and others. Below we briefly discuss the alternative-path as a recovery action as it highlights the need for a reasoning-based approach.

The recovery process is handled by adding the current process state (with the violation) and re-invoking the reasoner to perform abductive reasoning for the goal. However, it is not always possible to recover from a violation *AND* respecting the associated constraints and composition goal. As a result, some constraints may require to be relaxed and the proposed approach allows to identify the conflicting clauses and hard-constraint if a recovery solution is not possible. The proposed approach thus preserves all the functional and non-functional constraints associated with the composition process (unless needed to be relaxed) while performing recovery. Further, the proposed approach allows both to find a new solution based on the current process state (thus specifying what steps should be taken now to recover from the violation and hence termed forward recovery) or to backtrack to some previous activity (if possible) and try to find a new from there. The response phase may require the execution run-time support (for instance actions such as logging, KPI's measurement, see Figure 8.1–*3*➝*2*) and may also require the support from the DECReasoner in order to do abductive reasoning for actions such as finding alternatives (see Figure 8.1–*3*➝*1*).

8.4 Example

For the motivating example, we consider that the event-calculus based composition process specification has been instantiated, as discussed in Section-7.2.1, and different solutions and cases have been explored, as discussed in Section-7.3.5. One particular instantiated solution (or a set of solutions based on initial context) is then chosen for execution and for this example, we consider that the initial solution requiring CRR/FC/SC to be there, Section-7.2.1 is chosen for execution. The solution returned by the reasoner states what event *happen* at which time-point and corresponding world-state representing the effects of events. The event-calculus *Happens* axioms thus can serve as the plan to execute the process and the table below represents the execution plan for the motivating example:

Execution plan for the motivating example

```
Happens(Start_activityWS(Start), 0).
Happens(EvalCondTrue(IsCRRAvailable/IsFCAvailable/IsSCAvailable), 0).
Happens(EvalCondFalse(FalseAlarmCheckByCRR), 1).
Happens(Invoke(AmbulanceWS/CallStaffWS/FireBrigadeWS/MeteorologyDeptWS/PoliceWS), 2).
Happens(EvalCondTrue(HasFBArrived), 17).

Happens(EvalCondFalse(FalseAlarmCheckByFB), 18).
Happens(Start_activityS(FireContainment), 19).
Happens(End_activityS(FireContainment), 39).
Happens(Start_activityS(ExamineSite), 40).
...
```

According to execution plan shown above, at the start of process the availability of emergency staff is checked and thus as the result of an unfortunate fire incident once the fire-alarms are activated (or composition process needs to be started manually), the availability of emergency staff is checked. The execution plan further requires all CRR/SC/FC to be there however as mentioned earlier this plan is based on the solution returned by the reasoner for the design-level contracts. Let us consider that at the execution time, it is the case that only CRR is there and other emergency staff is absent. This would result in a mismatch from the execution plan and a violation is thus detected. In order to cater for the violation (and to get an updated execution plan) the process specification needs to be updated by adding the violation and re-invoking the reasoner. The updated composition process is shown below:

Updated process specification to cater for monitored violation

```
Process specification ...
Happens(EvalConditionTrue(IsCRRAvailable), 0).
Happens(EvalConditionFalse(IsFCAvailable), 0).
Happens(EvalConditionFalse(IsSCAvailable), 0).
```

Invoking the reasoner gives us a new solution (as similar to the one discussed in Section-7.3.5) suggesting that as the FC/SC are not available, they can still be there once the CallStaffWS is invoked. The solution re-computation to cater for violation to the execution plan is repeated for all the violations to the execution plan such as the time taken for the fire-containment, arrival time for the emergency staff and so on. Further at any time during process execution, the process can be updated to have the components and constraints to be added, modified and removed. For instance, let us consider a new requirement to be added to the process (once it is known that the FC/SC are not there) that the arrival time of staff members (SC/FC) should be logged. In order to add the logging monitoring property, we can define multiple activities for logging each staff member, such as LogSCArrival and LogFCArrival. Then we can add axiom that whenever the arrival condition for some staff member holds true, we start the corresponding logging activity. The event-calculus model below should be added to process specification for the logging, *activitywithoutstate* abbreviated as *activity_ WS*:

Process change while in execution - adding logging property

activity_ WS LogFCArrival, LogSCArrival
Happens(Start_ activityWS(Log(FC/SC)Arrival), time) → HoldsAt(CondTrue(Has(FC/SC) Arrived), time).

Then, the reasoner needs to be re-invoked to have an updated solution to cater for the newly added requirement and the result returned by the reasoner (shown below) shows that at the arrival of the FC and SC the corresponding logging activities need to be started.

Updated instantiated solution to cater for process change

...
27
Happens(EvalCondTrue(Has(FC/SC)Arrived), 27). 28
+ConditionTrue(HasFCArrived). +ConditionTrue(HasSCArrived).
Happens(Start_ activityWS(Log(FC/SC)Arrival), 28).
29
+Finished_ activityWS(Log(FC/SC)Arrival).
30
...

In order to further elaborate the process change while in execution we consider a new security requirement added to the composition process specification once the ExamineSite activity is started. It is required that the recovery process for the items should be only be handled by a professional conservator. In order to handle this requirement, we can create a new condition representing that the Conservator has been arrived (HasConservatorArrived) and lets assume that the conservator arrives 20 minutes after the ExternalOrgWS has been invoked. The process specification is updated as follows:

Process change while in execution

condition HasCONSArrived
Happens(EvalCondTrue(HasCONSArrived), time) | Happens(EvalCondFalse(HasCONSArrived), time) → HoldsAt(RespRecvd_synchservice(ExternalOrgWS),time).

Happens(Invoke_synchservice(ExternalOrgWS), time) → Happens(EvalCondTrue(HasCONSArrived), time+20) | Happens(EvalCondFalse(HasCONSArrived), time+20).
Happens(EvalCondTrue(HasCONSArrived), time) | Happens(EvalCondFalse(HasCONSArrived), time) → Happens(Invoke_synchservice(ExternalOrgWS), time-20).

Happens(Start_activityWS(RecoverPriorityItems),time) → HoldsAt(Finished_activityWS (PlanRecovery),time) & HoldsAt (CondTrue(HasCONSArrived),time).

The last axiom in the model above updates the invocation axiom for the RecoverPriorityItems activity and enforces that the conservator should be there before the start of the RecoverPriorityItems activity. Changing the process specification will again require re-instantiating the process and in order to get an updated solution we update the composition process to include the events that have already happened (the partial execution plan), and models for newly added security requirement and re-invoke the reasoner. The updated model is shown below:

Updated instantiated solution to cater for process change

...
Happens(EvalCondTrue(IsExternalHelpNeeded), 46).
47
+CondTrue(IsExternalHelpNeeded). Happens(Invoke(ExternalOrgWS), 47).
48
+ResponseReceived(ExternalOrgWS).
...
62
Happens(EvalCondTrue(HasCONSArrived), 62).
63
+CondTrue(HasCONSArrived). Happens(End_activityS(PlanRecovery), 63).
64
-Started_activityS(PlanRecovery). +Finished_activityS(PlanRecovery).
Happens(Start_activityS(RecoverPriorityItems), 64).
65
+Started_activityS(RecoverPriorityItems).
...
84
Happens(End_activityS(RecoverPriorityItems), 84).
85
-Started_activityS(RecoverPriorityItems). +Finished_activityS(RecoverPriorityItems).

8.5 Summary

In this chapter, we have discussed the proposed approach for monitoring Web services composition process while in execution. The proposed monitoring framework [Zahoor 2011] is event-based and allows to specify and reason about the monitoring properties during composition process execution. The proposed monitoring framework (see Figure 8.1-③) is divided into three phases and in this chapter we have first discussed the *specification* phase which requires the user to specify the functional and non-functional properties that needs to be monitored to identify anomalies or needed for KPI's measurement.

Further, we have discussed the *detection and effects calculation* phase which is both responsible for detecting any violations based on the specified properties and to calculate the side-effects the detected violation has on the overall process. Then, we have discussed the *response* phase which uses the user-specified actions to respond to the monitored property. We have also discussed the monitoring properties specification, detection, effects calculation and recovery for the motivating example.

Part V

EPILOGUE

CHAPTER 9

Implementation

Contents

The event-calculus models for the proposed framework are specified using the discrete event calculus language [Mueller 2006] and all the models mentioned earlier can be directly used for reasoning purposes. In this chapter we will first discuss overall architecture for the proposed approach. We will then present the ECWS tool and enhancement to the *DECReasoner* for process verification and optimizing the encoding process, in the sections to follow. In the last section, we will detail the performance evaluation results.

9.1 Overview

As discussed earlier, the *DECReasoner* is a program for performing automated commonsense reasoning using the discrete event calculus [Mueller 2006]. It supports different reasoning modes (as required by the proposed approach) including deduction, abduction, model finding and others. The tool is open-source and thus we were able to modify the source code to enhance the SAT encoding process and include *zchaff/zverify_ df* solvers support for the process verification.

The event-calculus based composition model and *DECReasoner* can reason about the Web services composition process at an abstract level. In order to have a concrete solution we propose the following implementation architecture, Figure-9.1. The composition process starts when the user specifies the composition design, using a Java based application called *ECWS*, allowing to drag and drop components and provide constraints. Then, the Java-based application translates the composition design to event calculus based model in following phases:

The *pre-processing* phase discovers and binds the Web service nodes to concrete Web services instances using the SQWRL based approach, we discussed in

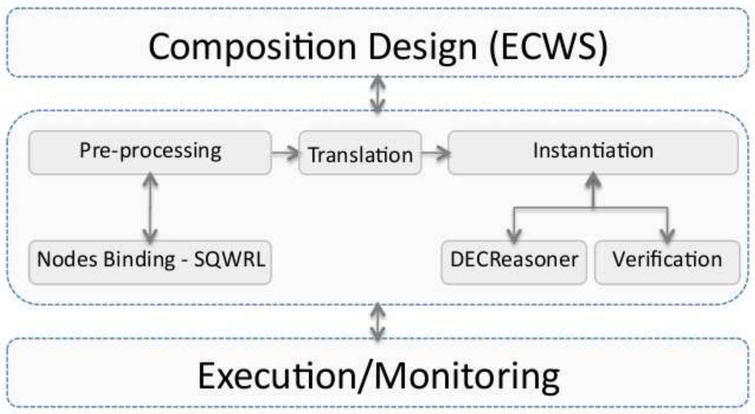


Figure 9.1: Implementation architecture

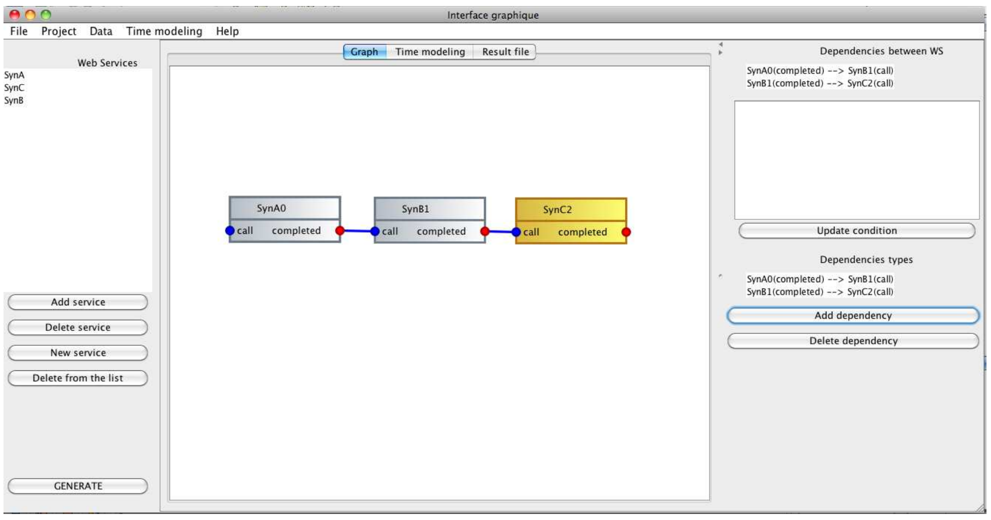
[Zahoor 2009b]. The nodes instantiation process itself can be purely event-calculus based as we discussed earlier, but due to performance issues we propose to use the pre-processing phase. The *pre-processing* phase may also involve handling the time-units of different participating components as we discussed in Section-6.1.5 and allowing the user to select one particular service from the candidate services returned as a result of nodes instantiation.

The *translation* phase follows which does the event calculus translation using the following guidelines. For all the services (and other components), corresponding include files are added to resulting output file (along with general files root.e and ec.e). Then for specifying control/data flow, temporal and security requirements corresponding patterns are used and resulting axioms are added to the event-calculus model.

The *DECReasoner* is then invoked to process the file, during the process *instantiation* phase, to either provide possible solutions or detect conflicts. In case of conflicts the *verification* phase attempts to identify the conflicts and may require to updated the composition design to resolve the conflicts or hard constraints and to re-invoke the reasoner. In case of multiple solutions, the user can be given option to select one particular solution which is then used by the Java application to perform the actual services execution. The semantics of different events (such as invoke) can be roughly mapped to WS-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.

9.2 Composition design using ECWS

In order to abstract the event-calculus 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, Figure-9.2. The application has three different sections, the left one listing components (Web services), the right one allows to specify the control/data flow amongst components and the middle one showing process specification and any results obtained.



SynA0(completed) --> SynB1(call)
SynB1(completed) --> SynC2(call)

SynA0(completed) --> SynB1(call)
SynB1(completed) --> SynC2(call)

Figure 9.2: The ECWS application for composition design

The components section includes the Web services already added and options to add or remove more services, Figure-9.3. Instances of already known services can be added to (or removed from) the process specification section and once added they can be dragged and moved around. The control/data flow section includes options to specify the dependency amongst services and update any conditions if conditional invocation is sought, Figure-9.3. The dependencies can be added and removed and specifying multiple services to be dependent on some particular service, will result in a Split and ECWS will show a dialog box allowing user to specify the Split scheme (AND/OR based split, Section-5.2). The process specification section is also updated to reflect any dependencies added between services.

Then on the bottom left of the components section, is the *GENERATE* button that will generate event-calculus models for the specification and will automatically invoke the *DECReasoner* to reason about the generated even-calculus 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 which shows the models using a graphical interface, Figure-9.4.

The proposed ECWS application for the process specification is still in early

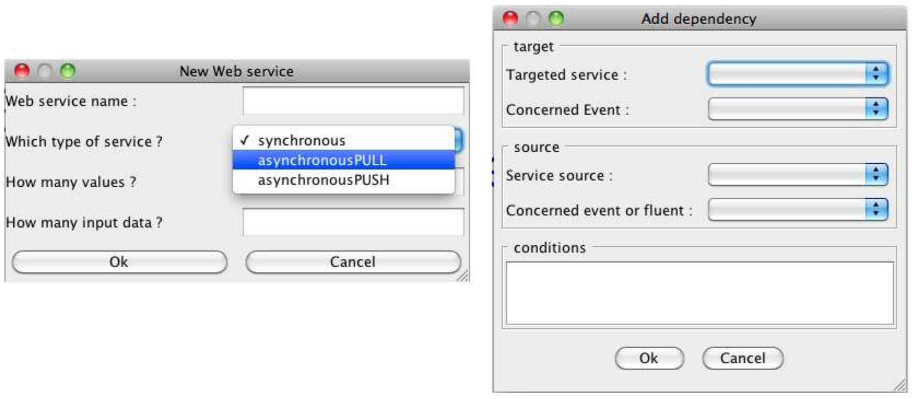
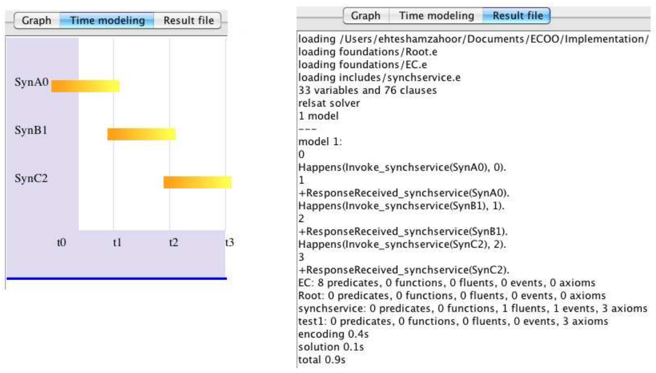


Figure 9.3: Adding a new service and specifying dependency between Web services

phases and only serves as a proof of concept prototype. Although it can handle partial process specification, can automatically generate event-calculus models and can directly invoke the reasoner and parse the results returned, it does not handle process verification and monitoring.



```
loading /Users/ehteshamzahoor/Documents/ECOO/Implementation/
loading foundations/Root.e
loading foundations/EC.e
loading includes/synchservice.e
33 variables and 76 clauses
relsat solver
1 model
---
model 1:
0
Happens(Invoke_synchservice(SynA0), 0).
1
+ResponseReceived_synchservice(SynA0).
Happens(Invoke_synchservice(SynB1), 1).
2
+ResponseReceived_synchservice(SynB1).
Happens(Invoke_synchservice(SynC2), 2).
3
+ResponseReceived_synchservice(SynC2).
EC: 8 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
synchservice: 0 predicates, 0 functions, 1 fluents, 1 events, 3 axioms
test1: 0 predicates, 0 functions, 0 fluents, 0 events, 3 axioms
encoding 0.4s
solution 0.1s
total 0.9s
```

Figure 9.4: Displaying result after invoking *DECReasoner*

9.3 Enhancements to *DECReasoner*

As discussed earlier, in order to use the *DECReasoner* domain description (specified using Discrete Event Calculus Reasoner language) that includes an axiomatization describing domains of interest, observations of world properties at various times, and a narrative of known event occurrences is placed in a file. The *DECReasoner* is then invoked for the domain description and it firsts transforms the domain description into a satisfiability (SAT) problem. It then invokes a SAT solver (relsat), which produces zero or more solutions and resulting solutions are decoded and displayed to the user.

9.3.1 Process verification using *zchaff/zverify_ df*

However, if *relsat* solver produces no models (as a result of some conflict in the process specification), the *DECReasoner* then invokes the *walksat* solver first with the *-target* parameter having value 1 and then (if previous run fails) with *-target* parameter having 2. If the *walksat* run fails again, the *DECReasoner* gives up without providing any solution. If the invocation of *walksat* solver does return a set of unsatisfiable clauses it is suggested that one or two *unsatisfied* clauses may be helpful for debugging while three or more unsatisfied clauses tend to be less useful [Mueller 2006]. Further it is suggested that as the *walksat* is stochastic, it is possible to get a different set of unsatisfiable clauses on different runs and by looking at different sets of unsatisfiable clauses would be helpful in debugging the process model.

However, we believe that this approach may be somewhat helpful for simpler models however it is not possible to use this as a verification approach as neither *walksat* is always able to identify unsatisfiable core, nor the size of unsatisfiable core is small enough to be manually observed. As a result we have modified the *DECReasoner* code to have *zchaff/zverify_ df* as the solver to use for the process verification. *Zchaff* is an implementation of the *Chaff* algorithm and won the Best Complete Solver in both industrial and handmade bench-mark categories for SAT 2002 Competition. It is also integrated with the BlackBox AI planner, NuSMV model checker and others. The output from *zchaff* can also be used by *zverify_ df* tool to get a set of unsatisfiable clauses that can be used for the process verification.

The modified verification approach thus first invokes the *zchaff* solver instead of *walksat* and the output from the *zchaff* is passed to *zverify_ df* for identifying a set of unsatisfiable clauses. These unsatisfiable clauses can be then filtered, as discussed in Section-7.3.4 and we have also modified the encoding process to log that which clauses represent the frame axioms (and others) and this information can help to automate the filtering process.

9.3.2 Event-calculus to SAT encoding

Then, one important limitation of *DECReasoner* is the time taken for event-calculus to SAT encoding which increases exponentially with the increase in time-points and introducing complex axioms involving multiple free variables, as we discussed in [Zahoor 2010a, Zahoor 2010b]. As the proposed approach requires invoking the reasoner multiple times (instantiation, monitoring and recovery) the encoding process can thus be bottleneck. 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 (using *Shark*) helped us identify that the *strcmp* function is proving to be bottleneck once an element is sought from the hash table. On further investigation, 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 we also discussed the proposed changes with Erik T. Mueller, who is responsible for initial implementation of *DECReasoner* at IBM and the changes can now be downloaded from *DECReasoner* official Website [1].

```
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);
}
```

9.4 Performance evaluation

In this section we will detail the performance evaluation results for the motivating example. 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. In order to complicate the composition process to highlight the performance of different solvers for process instantiation/verification and the effect of proposed modifications to the event-calculus to SAT encoding process, we consider two cases.

The first one further complicates the motivating 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

[1] http://decreasoner.sourceforge.net/

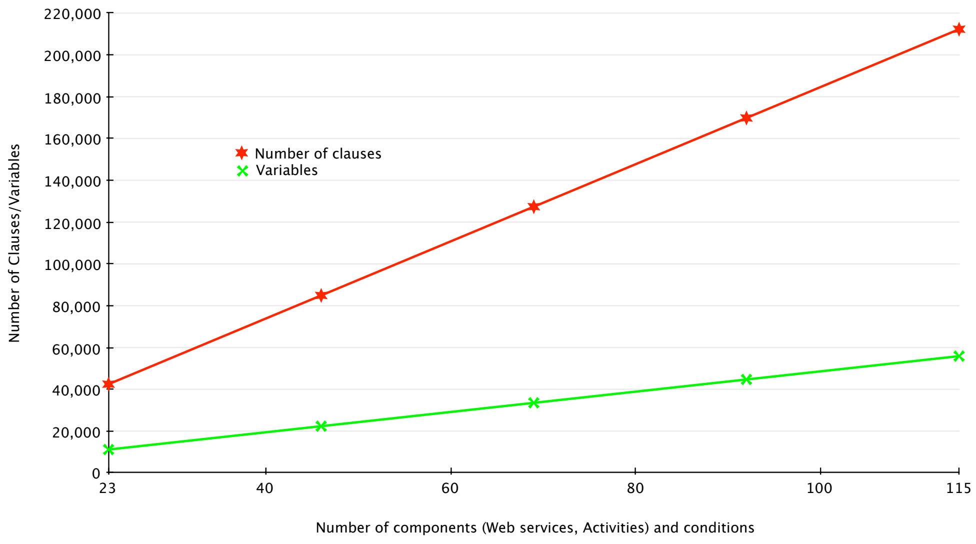


Figure 9.5: Problem size increase with the number of components

(and conditions). The corresponding increase in the problem complexity and state space (as we discussed for motivating the use of SAT-solver for process verification in Section-7.3.1) is evident from Figure-9.5, with Y-axis showing the number of variables and clauses in the SAT encoding for the motivating example while the X-axis showing the number of components.

The performance evaluation results, Figure-9.5 show that the increase in number of components results in increase in problem size and so as the state space needed to model check the composition process. Further, we consider the time taken for process instantiation (solution computation using relsat solver) and the effect of proposed modifications to the encoding process in Figure-9.6, with X-axis representing the number of components and the Y-axis representing the time in seconds. The performance results indicate that the motivating example requires 7.6 seconds for encoding the problem into a SAT problem (the original encoding approach) and only 0.3 seconds for solution finding using relsat solver. However, the original encoding process does not scale well with the increase in problem size (by adding components) while the modified encoding process (based on the changes we proposed in Section-9.3.2) performs considerably better. The solution computation using the relsat is very efficient.

Next, we consider the performance evaluation results for the process verification using zchaff/zverify_df solver, Figure-9.6 with X-axis representing the number of components and the Y-axis representing the time in seconds. The results indicate the zchaff/zverify_df to be very efficient but this observation stems from the fact that the zchaff/zverify_df only tries to find the minimal unsatisfiable-core, which stays small even with the addition of more-components.

The composition process can also be complicated by adding the number of

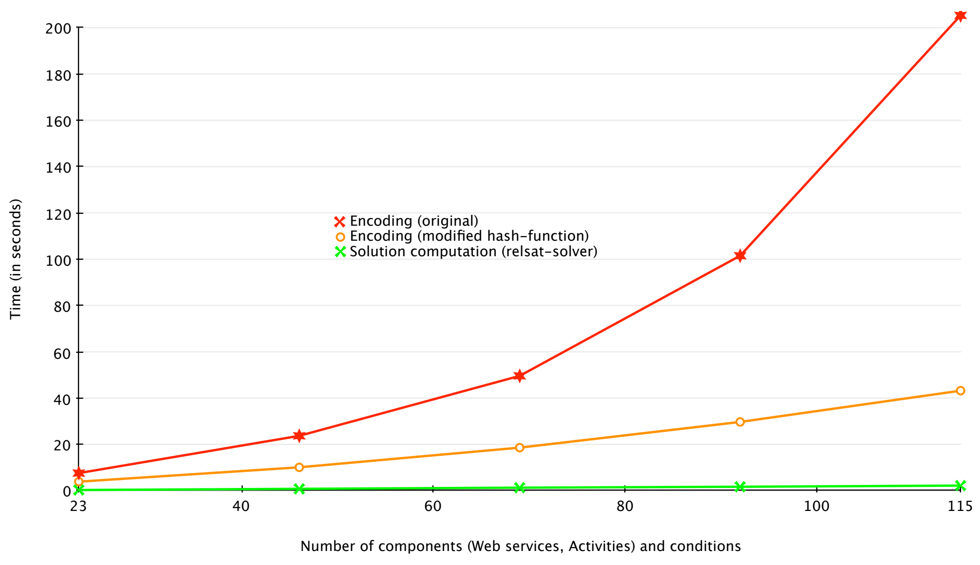


Figure 9.6: Encoding/solution time with increase in number of components

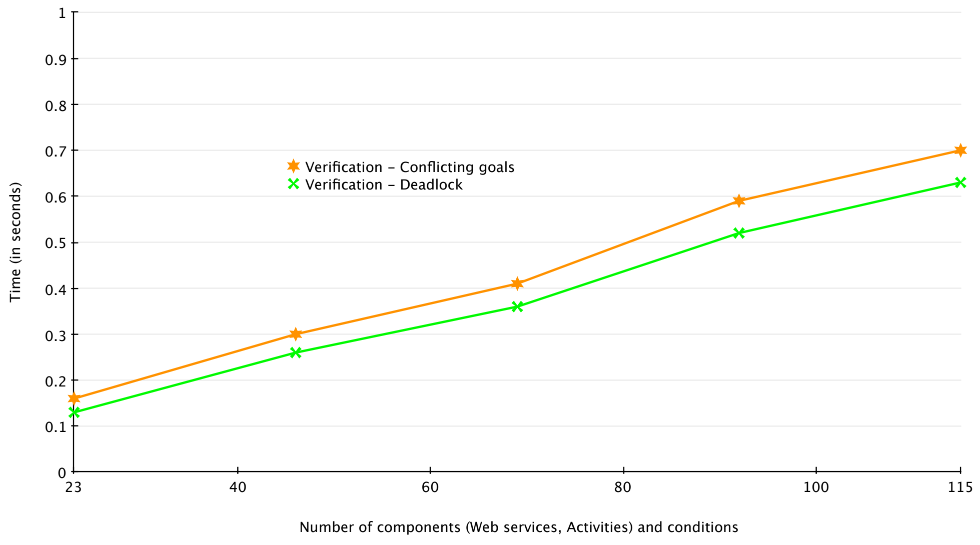


Figure 9.7: Unsatisfiable-core computation using zchaff/zverify_df

time-points for the process specification and next we consider the performance evaluation results for the motivating example, with the increase in time-points. As with the case with the increase in number of components, the performance evaluation results, Figure-9.8, indicate that the increase in number of time-points results in increase in problem size and so as the state space needed to model check the composition process.

The performance evaluation results for the time taken for process instantiation (solution computation using relsat solver) and the effect of proposed modifications

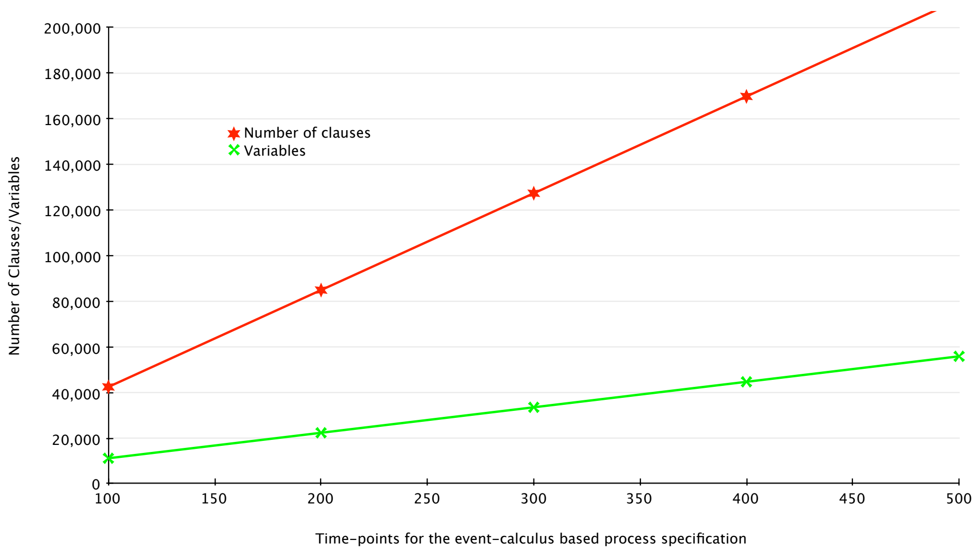


Figure 9.8: Problem size increase with the number of time-points

to the encoding process, with increase in the time-points are shown in in Figure-9.9, with X-axis representing the time-points and the Y-axis representing the time in seconds. The performance results indicate (as similar to Figure-9.6) that the the modified encoding process (based on the changes we proposed in Section-9.3.2) performs considerably better than the original encoding process and again the solution computation using the relsat is very efficient.

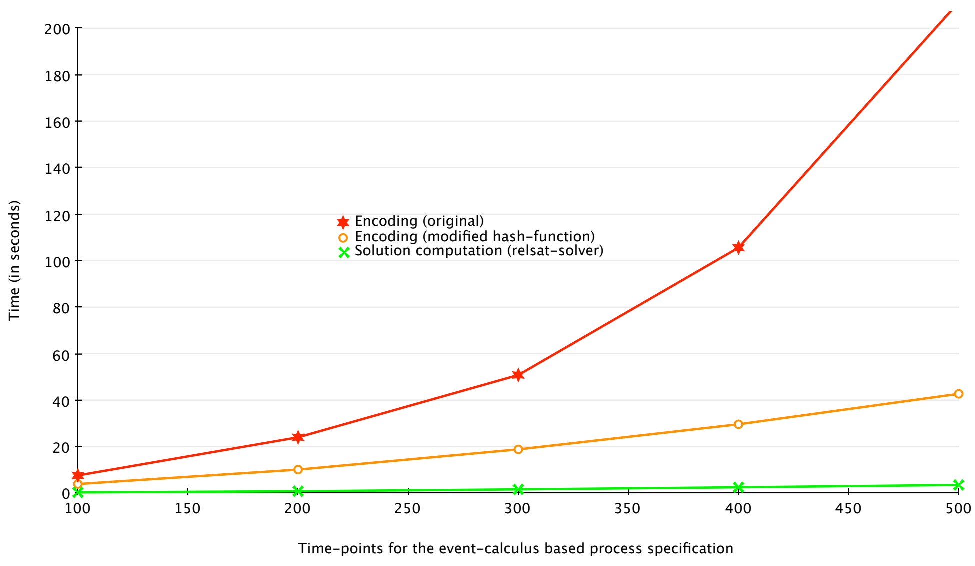


Figure 9.9: Encoding/solution time with increase in number of time-points

Chapter 10

Conclusion

Contents

In this thesis, we have presented an event-based declarative and integrated approach called **DISC** that aims to bridge the gap between composition design, verification and monitoring. In this chapter will first review the problem description and then will briefly discuss the proposed approach focusing on use of event-calculus for specifying composition design, and the proposed approach for design-time verification and execution time monitoring and recovery. Further, we will discuss the known limitations of the proposed approach and the future perspectives.

10.1 Problem description

Web services are defined as the software systems designed to support interoperable machine-to-machine interaction over a network[1] and they are in the mainstream of information technology, paving way for inter and across organizational application integration. Individual services may need to be composed to satisfy user needs and thus 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

[1]http://www.w3.org/TR/ws-gloss/

conflicts (such as deadlocks) in the process specification and once the process has been verified it is executed. Further, as the Web services are autonomous and only expose their interfaces, composition process is based on design level service contracts and the actual execution of composition process may result in the violation of the design-level services contracts due to errors such as network or service failures, change in implementation or other unforeseen situation. This highlights the need to monitor and detect the errors and react accordingly to cater for them.

The significance of Web services and composition of Web services into value-added process has led to a number of proposed approaches that aim to handle different aspects related to Web services composition at different life-cycle stages. However, the proposed approaches are procedural (and rigid to handle dynamically changing situations) and focus on some stage(s) of the composition process life-cycle. 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 thus mark the motivation for our work. Below we briefly discuss the limitations of the proposed approaches:

Lack of integration - The traditional approaches focus only on some stages of process life-cycle and this lack of integration results in a complex model such as at design-time first using *BPMN/WS-BPEL* for the process specification. Then transforming the process specification to a particular automata with guards, and using SPIN model checker [Fu 2004] for process verification, and finally using some event-based approach for process monitoring while in execution. The lack of integration not only results in a complex model but it is not always possible to have a complete transformation from one modeling approach to other and with the addition of non-functional (such as security and temporal) requirements the transformation becomes even more complex and challenging. In addition, the lack of integration leads to the approaches that does not allow to learn from the run-time failures to provide the recovery actions such as re-planning.

Procedural composition model - A process model is called procedural when it contains explicit and complete information about the flow of the process but only implicitly keeps track of why these design choices have been made and if they are indeed part of the requirements or merely assumed for specifying the process flow [Goedertier 2008]. Although this adds a lot to the control over the composition process, however as there is tradeoff between the control and flexibility, this control comes at the expense of process flexibility and thus making the process rigid to adapt to continuously changing situations and possibly not even conforming to the process specification requirements. On the other hand, a process is termed as declarative when it models the minimal (and only specified) requirements that mark the boundary of the process and any suitable execution plan which meets the specified requirements is sought.

The proposed graph-based composition modeling approaches are mostly procedural and although the graph-based approaches tend to be simpler and intuitive for the process modeler for the composition design, they over-constrain the process assuming the design choices that may not be present in the requirements but only added to specify the process flow. The paradigm change from procedural to declarative process modeling was advocated in [Pesic 2006] by introducing the ConDec language for declarative process modeling. However, we believe that the traditional AI planning (and so as rule-based) approaches for composition process modeling are declarative and they have advantages in terms of expressibility, flexibility and adaptability and dynamism as they are more expressive based and are based on formal logic and flexible as only constraints that mark the boundary of the process are specified. Further, the declarative approaches allow for specifying more workflow patterns than the procedural ones.

Verification - The proposed approaches for the composition verification, in general, require mapping the process (mostly defined using procedural approaches such as BPEL) to some formal logic (such as petri-nets, automata or process logic) and then using model checkers to verify the composition process. This transformation based approach has two major limitations, first the proposed verification approaches are based on traditional procedural approaches and as we detailed in the Chapter-3, procedural approaches have less expressibility, flexibility and adaptability and dynamism as compared to the declarative ones. Further, the limited expressibility makes it difficult to verify the non-functional properties (such as temporal, security requirements or more importantly their combinations) associated with the composition process as for first, it is difficult to specify the non-functional properties using the traditional approaches such as BPEL and a number of approaches have been proposed as an extension to BPEL for specifying non-functional aspects and then it is not trivial (is possible) to add formal semantics to them for their verification.

Specifying the exact and complete sequence of activities to be performed for the composition process, as required by the traditional procedural approaches, however does make it possible to use proposed automata or petri-nets based approaches for design-time verification of composition process, however with the declaratives approaches the process may be only partially defined and thus this makes it difficult to use traditional approaches for the process verification as the transition system for a declarative process can be very large. Further, the design-time verification should be coupled with execution-time monitoring and complexity of these approaches make them difficult to use for verifying the functional and non-functional constraints associated with the process while handle process change or recovery.

Event-based monitoring - Traditional approaches for the composition monitoring are proposed as an extension to some particular run-time and are tightly coupled and limited to it. In contrast the use of an event-based approach works on the message-level and thus is unobtrusive, independent of run-time and allows for integration of other systems and processes, as discussed in [Moser 2010]. Then, the traditional monitoring approaches [Barbon 2006a, Baresi 2009, Mahbub 2004] build upon composition frameworks that are highly procedural, such as BPEL, and this in-turn poses two major limitations. First, they limit the benefits of any event-based monitoring approach as the events are not part of the composition framework and functional and non-functional properties are not expressed in terms of events and their effects. Secondly the use of procedural approach for process specification does not bridge the gap between organization and situation in a way that it is very difficult to learn from run-time violations and to change the process instance (or more importantly process model) at execution time, and it does not allow for a reasoning approach allowing for effects calculation and recovery actions such as re-planning or alternate path finding as we discussed in [Zahoor 2010a].

10.2 The proposed approach

The motivation of our work stems from the process modeling, design-time verification, execution-time monitoring and recovery in an integrated and declarative way to cater for dynamically changing situations (for instance, crisis handling or the logistics processes) and we have presented a real-world crisis handling process in Section-2.2.

The objective of our thesis is thus to handle the process modeling, design-time verification, execution-time monitoring and recovery in an **integrated** and **declarative** way. Declarative approach results in a highly flexible composition process that may be needed to cater for dynamically changing situations while integration simplifies the approach by using the similar formalism for composition design, verification and monitoring. The use of declarative and integrated approach further allows to have recovery actions such as re-planning (to cater for monitored violations during process execution) which are difficult to achieve using traditional approaches. In this thesis, we have proposed an integrated 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. In the sections to follow, we will briefly discuss the proposed approach.

10.3 Declarative composition design

The composition process modeling is the first and most important stage of the composition process life cycle. The objective of composition process modeling is to provide high-level specification independent from its implementation that should be easily understandable.

10.3. Declarative composition design

The proposed framework allows for a composition design that is declarative and 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. We have based the composition design on event-calculus and defined patterns for specifying the functional and non-functional aspects using event-calculus for process specification, instead on relying on different formalisms or extensions for specifying different aspects as required by traditional approaches such as WS-BPEL. In this section we will first provide an overview of different components modeling using event-calculus. Further, we will discuss patterns for specifying control and data flow between them and will also provide an overview for modeling non functional (temporal and security) requirements modeling using event-calculus.

10.3.1 Components

The various components that constitute the composition design can be broadly divided into **activity** and **service** categories, Section-4. Activity is a general terms for any work being performed while the services include either the Web services instances already known or abstract Web services (called nodes) that need to be discovered based on some specified constraints. Detailed event-calculus based models for different activity types can be found in Section-4.1 and the table below provides pointers to different related sections for modeling activities.

Activities	***Section for event-calculus models***
Activities with states	*Event-calculus model is discussed in Section-4.1.1, in order to use the model include activitywithstate.e to the event-calculus file.*
Instantiated model	*A basic example showing how to use the activitywithstate.e for reasoning using DECReasoner is shown in Section-4.1.2.*
Without states	*A simplified model, where activity states are not needed is discussed in Section-4.1.3 and corresponding include file is activitywithoutstate.e.*
Activities with restart	*Activities that need to be restarted are modeled in Section-4.1.4 and corresponding include files are activity(with/without)staterestart.e*

Then, detailed event-calculus models for Web services supporting different invocation modes (synchronous, asynchronous) can be found in Section-4.2 and the table below provides pointers to different related sections for modeling different Web service types.

Web services	***Section for event-calculus models***
Synchronous	*Web services model with synchronous invocation mode is discussed in Section-4.2.1. Corresponding include files are synchservice(withdelay).e.*
Asynchronous (pull)	*Pull-based asynchronous Web services invocation model is presented in Section-4.2.2, corresponding include file is asynchpullservice.e.*
Asynchronous (push)	*Push-based asynchronous invocation model is discussed in Section-4.2.3, corresponding include file is asynchpushservice.e.*
Services(re-invoke)	*EC model for Web services that need to reinvoked (for instance within loop body) is discussed in Section-4.2.4, corresponding include files are (asynchpull/asynchpush/synch)servicewithreinvoke.e*
Nodes	*A brief discussion about the nodes is presented in Section-4.3.*

10.3.2 Control/Data flow specification

We have presented a pattern-based approach for specifying the control/data flow between different components and detailed discussion about patterns and their usage for different constructs can be found in Section-5. The table below provides pointers to different related sections for modeling different control and data flow constructs.

Construct	***Section for event-calculus model***
Dependency	*Event-calculus based patterns for specifying dependency between two components is discussed in Section-5.1.*
Split & Join	*Patterns for Split and Join constructs (including different split/join schemes) is presented in Section-5.2.*
Conditions	*Event-calculus model for specifying conditions is discussed in Section-5.3 and corresponding include file is condition.e.*
Iteration	*Patterns for specifying iteration construct is discussed in Section-5.4.*
Data	*Event-calculus model for specifying request/response data is discussed in Section-5.5, corresponding files are request.e and response.e.*
Messages	*EC model for specifying the message transfer between two components including patterns for its usage is discussed in Section-5.6, corresponding include file is messagedata.e.*

10.3.3 Temporal and security aspects

Using event-calculus as the modeling formalism also allows to specify the temporal constraints for the composition process, Section-6.1. The temporal constraints can either be local to a component (such as specifying the time it takes for a service to produce response) or they can be imposed by the composition process on the participating components (such as specifying the delay between the invocation of two services). Further, the temporal constraints can be either control based (such as specifying the delay between the successive invocation of component) or they can be data-based temporal constraints (such as data validity constraints). The table below provides pointers to different related sections for modeling different patterns of temporal requirements.

Temporal patterns	***Section for event-calculus model***
Response time	*Event-calculus based pattern for specifying the time taken by components to finish execution is discussed in Section-6.1.1.*
Restart/Refresh	*Pattern for specifying components re-invocation (either after some specific time or based on data invalidity) is presented in Section-6.1.2.*
Time frame	*EC Pattern for specifying the time frame allowed for a component's execution is presented in Section-6.1.3.*
Interval algebra	*Patterns based on base relations of Allen's interval algebra are discussed in Section-6.1.4.*
Time units	*An approach to cater different time-units used for specifying temporal aspects for different components is discussed in Section-6.1.5.*

The choice of event-calculus as the modeling formalism also allows to handle the security requirements in the composition process. An overview about different security requirements for the composition process can be found in Section-6.2.1. Further, the security requirements can be handled at different interaction levels for the participating Web services in the composition process. The table below provides pointers to different related sections for modeling different security requirements.

Security requirement	***Section for event-calculus model***
Data confidentiality, retention and integrity	*Event-calculus based patterns and brief for specifying data confidentiality, retention and integrity is discussed in Section-6.2.3.*
Authentication/ Authorization	*Patterns for specifying Authentication/Authorization is presented in Section-6.2.4.*
Dynamic Task Delegation	*A brief overview of the event-calculus based approach for dynamic task delegation is discussed in Section-6.2.5.*

10.4 Process verification and monitoring

For the process design-time verification we have proposed a symbolic model checking approach using satisfiability reasoning. The need for the satisfiability solving for process verification stems from multiple sources; first as the composition process may be declarative and partially defined by only specifying the constraints that mark the boundary of the solution to the composition process and the objective is to find solution(s) that respect those constraints (and which is connectivity verification property), satisfiability solving can thus be used to solve the problem by encoding it as a satisfiability problem, representing the constraints.

Further, the state space of a declarative process can be significantly large, as the process is only partially defined and all the transitions may not have been explicitly defined (in contrast to procedural approaches), and thus it makes it easier to do the symbolic model checking instead of using explicit representation of state transition graphs and/or using the binary decision diagrams. The verification properties can include the connectivity, compatibility and behavioral correctness (safety and liveness properties) and the proposed approach allows for both model checking the verification properties and for identifying and resolving the conflicts in the process specifications a result of process verification. Further, as the conflict clauses returned by the SAT solver can be very large, we have proposed filtering criteria to reduce the clauses and defined patterns for identifying the nature of conflicts.

For the execution-time monitoring (and recovery from any monitored violations) we have proposed an event-based message-level monitoring approach that allows to reason about the events and does not require to define and extract events from process specification, as the events are first class objects of both design and monitoring frameworks. As the proposed monitoring approach builds upon event-calculus based composition design, it allows for the specification of monitoring properties that are based on both functional and non-functional (such as temporal, security or their combinations) requirements. These properties are expressed as event-calculus axioms and can be added to the process specification both during process design and during the process execution. The proposed monitoring approach both allows for KPI's measurement(that may be needed for process evaluation or result in proactive detection of any violations) and the detection of violations once they happen. Different levels of detection are provided such as detection to the process execution plan, detection to the violations based-on any properties and events added during process execution and others.

Further, the web services composition problem is traditionally 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 process execution. 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. This also allows to cater for the "ripple effect" any violation has on the process execution, and for proactive detection of any possible violation that is bound to happen later in the process execution, as a result of current detected violation. In addition, once a violation is detected and a recovery solution (for instance re-planning) is sought, the proposed approach allows both to find a new solution based on the current process state (thus specifying what steps should be taken now to recover from the violation and hence termed forward recovery) or to backtrack to some previous activity (if possible) and try to find a new from there. Then, any recovery solution takes care of the functional and non-functional properties associated with the process, when performing recovery.

10.5 Implementation architecture

The proposed event-calculus models presented in this work are mentioned using the discrete event calculus language [Mueller 2006] and they can be directly used for reasoning purposes. We have proposed an implementation architecture and implemented a Java-based application that allows to abstract the event-calculus models to the end-user. It facilitates the composition design process by providing a user-friendly interface for specifying composition design (including entities and control/data flow between them) and allows to automatically generate the corresponding event-calculus models, invokes the reasoner and shows the results returned by the reasoner.

The proposed approach uses the DECReasoner as the event-calculus reasoner, however as we discussed in [Zahoor 2010a, Zahoor 2010b] the event-calculus to SAT encoding process provided by the reasoner, does not scale well. We have thus modified the DECReasoner code to gain substantial performance improvement as evident in performance evaluation results, Section-9.4. Further, we have presented a real world crisis management case-study and discussed how a process-based approach can be beneficial. For process verification, we extended *DECReasoner* [Mueller 2006] 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.

10.6 Perspectives and limitations

Regarding the limitations of the approach presented in this thesis, one observation is regarding the abstraction level chosen for modeling Web services and activities. We have extensively modeled different components at an abstract level and although the models presented are very expressive, they may need to be modified and updated for modeling some concrete low-level details (if needed to be reasoned about). As the proposed approach is extensible and is based on expressive event-calculus, the proposed models can thus be modified and new ones can be added to handle other requirements.

Then, the proposed ECWS tool for the process specification is in early phases and only serves as a proof of concept prototype. Although it can handle partial process specification, can automatically generate event-calculus models and can directly invoke the reasoner and parse the results returned, it does not handle process verification and monitoring. Further, regarding the changes to the DECReasoner, the modified event-calculus to SAT encoding does improve the time taken for encoding (as discussed in Section-9.3.2 and as evident from the performance evaluation results, Section-9.4) however the encoding can further be improved by calculating the hash values based on both first and last characters of the input symbol. Regarding the process verification, we have modified the DECReasoner to use zchaff/zverify as the solver to use when performing verification, however the current implementation should be modified by adding an option to *DECReasoner* to override the default use of *Walksat* solver. Further, the proposed filtering approach has not yet been automated and requires manually matching the process logs with conflicts clauses to filter the unsatisfiable core.

Regarding the future perspectives, we would like to have a complete implementation for the ECWS tool for declarative process specification, verification and monitoring. Further, the proposed models can be updated to cover other aspects regarding the services composition, such as compatibility analysis and automatic Web services composition. The use of event-calculus as the modeling formalism not only allows for expressiveness to model different related concepts but also allows to combine them, such as we mentioned in the thesis that the security and temporal aspects can be combined to have for instance, access control policies defined for specific time intervals as needed for dynamic task delegation. Thus any extension to the models presented in this work will allow to use the base models for specifying different functional and non-functional properties. The use of a pattern-based approach, where different event-calculus models are organized into self-contained and independent files, further aids to reuse and extend the proposed models.

Bibliography

[Ardagna 2007] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici and Pierluigi Plebani. *PAWS: A Framework for Executing Adaptive Web-Service Processes.* IEEE Software, vol. 24, no. 6, 2007. (Cited on pages 45 and 48.)

[Bandara 2003] Arosha K Bandara, Emil C Lupu and Alessandra Russo. *Using Event Calculus to Formalise Policy Specification and Analysis.* Policies for Distributed Systems and Networks, IEEE International Workshop on, vol. 0, page 26, 2003. (Cited on pages 6 and 86.)

[Barbon 2006a] Fabio Barbon, Paolo Traverso, Marco Pistore and Michele Trainotti. *Run-Time Monitoring of Instances and Classes of Web Service Compositions.* In ICWS, pages 63–71, 2006. (Cited on pages 26, 44, 47 and 134.)

[Barbon 2006b] Fabio Barbon, Paolo Traverso, Marco Pistore and Michele Trainotti. *Run-Time Monitoring of Instances and Classes of Web Service Compositions.* In ICWS, pages 63–71, 2006. (Cited on page 43.)

[Baresi 2005] Luciano Baresi and Sam Guinea. *Dynamo: Dynamic Monitoring of WS-BPEL Processes.* In ICSOC, pages 478–483, 2005. (Cited on page 43.)

[Baresi 2007] Luciano Baresi and Sam Guinea. *Dynamo and Self-Healing BPEL Compositions.* In ICSE Companion, pages 69–70, 2007. (Cited on page 44.)

[Baresi 2009] Luciano Baresi, Sam Guinea, Marco Pistore and Michele Trainotti. *Dynamo + Astro: An Integrated Approach for BPEL Monitoring.* ICWS, pages 230–237, 2009. (Cited on pages 26, 44, 47 and 134.)

[Baresi 2010] Luciano Baresi, Sam Guinea, Olivier Nano and George Spanoudakis. *Comprehensive Monitoring of BPEL Processes.* IEEE Internet Computing, vol. 14, no. 3, pages 50–57, 2010. (Cited on page 43.)

[Baresi 2011] Luciano Baresi and Sam Guinea. *Self-Supervising BPEL Processes.* IEEE Trans. Software Eng., vol. 37, no. 2, pages 247–263, 2011. (Cited on page 43.)

[Basin 2006] David A. Basin, Jürgen Doser and Torsten Lodderstedt. *Model driven security: From UML models to access control infrastructures.* ACM Trans. Softw. Eng. Methodol., vol. 15, no. 1, 2006. (Cited on page 36.)

[Beeri 2008] Catriel Beeri, Anat Eyal, Tova Milo and Alon Pilberg. *BP-Mon: query-based monitoring of BPEL business processes.* SIGMOD Record, vol. 37, no. 1, pages 21–24, 2008. (Cited on page 43.)

[Benatallah 2005] Boualem Benatallah, Fabio Casati, Julien Ponge and Farouk Toumani. *On Temporal Abstractions of Web Service Protocols.* In CAiSE Short Paper Proceedings, 2005. (Cited on page 36.)

[Berardi 2003] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini and Massimo Mecella. *e-Service Composition by Description Logics Based Reasoning.* In Description Logics, 2003. (Cited on page 32.)

[Blum 1997] Avrim Blum and Merrick L. Furst. *Fast Planning Through Planning Graph Analysis.* Artif. Intell., vol. 90, no. 1-2, pages 281–300, 1997. (Cited on page 32.)

[Bonet 2001] Blai Bonet and Hector Geffner. *Heuristic Search Planner 2.0.* AI Magazine, vol. 22, no. 3, pages 77–80, 2001. (Cited on page 32.)

[Bordeaux 2004a] Lucas Bordeaux, Gwen Salaün, Daniela Berardi and Massimo Mecella. *When are Two Web Services Compatible?* In TES, pages 15–28, 2004. (Cited on page 36.)

[Bordeaux 2004b] Lucas Bordeaux, Gwen Salaün, Daniela Berardi and Massimo Mecella. *When are Two Web Services Compatible?* In TES, pages 15–28, 2004. (Cited on page 41.)

[Cao 2005] Jiannong Cao, Jin Yang, Wai Ting Chan and Cheng-Zhong Xu. *Exception Handling in Distributed Workflow Systems Using Mobile Agents.* In ICEBE, pages 48–55, 2005. (Cited on page 44.)

[Casati 1999] Fabio Casati, Maria Grazia Fugini and Isabelle Mirbel. *An Environment for Designing Exceptions in Workflows.* Inf. Syst., vol. 24, no. 3, pages 255–273, 1999. (Cited on page 44.)

[Casati 2000a] Fabio Casati, Silvana Castano, Maria Grazia Fugini, Isabelle Mirbel and Barbara Pernici. *Using Patterns to Design Rules in Workflows.* IEEE Trans. Software Eng., vol. 26, no. 8, pages 760–785, 2000. (Cited on page 44.)

[Casati 2000b] Fabio Casati, Ski Ilnicki, Li-jie Jin, Vasudev Krishnamoorthy and Ming-Chien Shan. *Adaptive and Dynamic Service Composition in eFlow.* In Proceedings of the 12th International Conference on Advanced Information Systems Engineering, CAiSE '00, pages 13–31, London, UK, 2000. Springer-Verlag. (Cited on page 31.)

[Castilho 1999] Marcos A. Castilho, Olivier Gasquet and Andreas Herzig. *Formalizing Action and Change in Modal Logic I: the frame problem.* J. Log. Comput., vol. 9, no. 5, pages 701–735, 1999. (Cited on page 32.)

[Chollet 2008] Stéphanie Chollet and Philippe Lalanda. *Security Specification at Process Level.* In IEEE SCC (1), pages 165–172, 2008. (Cited on page 36.)

[Cicekli 2000] Nihan Kesim Cicekli and Yakup Yildirim. *Formalizing Workflows Using the Event Calculus.* In DEXA, 2000. (Cited on page 6.)

[Colombo 2006] Massimiliano Colombo, Elisabetta Di Nitto and Marco Mauri. *SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules.* In ICSOC, pages 191–202, 2006. (Cited on page 44.)

[Díaz 2005] Gregorio Díaz, Juan José Pardo, María-Emilia Cambronero, Valentin Valero and Fernando Cuartero. *Automatic Translation of WS-CDL Choreographies to Timed Automata.* In EPEW/WS-FM, pages 230–242, 2005. (Cited on page 39.)

[Dijkman 2007] R.M. Dijkman, M. Dumas and C. Ouyang. *Formal semantics and analysis of BPMN process models.* Rapport technique, Technical Report Preprint 7115, Queensland University of Technology, 2007. (Cited on page 40.)

[Dobson 2005] G. Dobson, R. Lock and Ian Sommerville. *QoSOnt: a QoS Ontology for Service-Centric Systems.* In EUROMICRO-SEAA, pages 80–87, 2005. (Cited on page 97.)

[Dong 2006] Jin Song Dong, Yang Liu 0003, Jun Sun 0001 and Xian Zhang. *Verification of Computation Orchestration Via Timed Automata.* In ICFEM, pages 226–245, 2006. (Cited on page 39.)

[Ellis 1993] Clarence A. Ellis and Gary J. Nutt. *Modeling and Enactment of Workflow Systems.* In Application and Theory of Petri Nets, pages 1–16, 1993. (Cited on pages 30 and 40.)

[Erol 1994] Kutluhan Erol, James A. Hendler and Dana S. Nau. *UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning.* In AIPS, pages 249–254, 1994. (Cited on page 34.)

[Fdhila 2008] Walid Fdhila, Mohsen Rouached and Claude Godart. *Communications Semantics for WSBPEL Processes.* In ICWS, pages 185–194, 2008. (Cited on page 6.)

[Ferrara 2004] Andrea Ferrara. *Web Services: A Process Algebra Approach.* CoRR, vol. cs.AI/0406055, 2004. (Cited on page 41.)

[Fikes 1971] Richard Fikes and Nils J. Nilsson. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.* Artif. Intell., vol. 2, no. 3/4, pages 189–208, 1971. (Cited on page 32.)

[Fox 2011] Maria Fox and Derek Long. *Efficient Implementation of the Plan Graph in STAN.* CoRR, vol. abs/1105.5457, 2011. (Cited on page 32.)

[Friedrich 2010] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici and Gaston Tagni. *Exception Handling for Repair in Service-Based Processes.* IEEE Trans. Software Eng., vol. 36, no. 2, pages 198–215, 2010. (Cited on pages 45 and 48.)

[Fu 2004] Xiang Fu, Tevfik Bultan and Jianwen Su. *Analysis of interacting BPEL web services.* In WWW, pages 621–630, 2004. (Cited on pages 5, 24, 39 and 132.)

[Gaaloul 2010] Khaled Gaaloul, Ehtesham Zahoor, Francois Charoy and Claude Godart. *Dynamic Authorisation Policies for Event-based Task Delegation.* In accepted in CAiSE, 2010. (Cited on pages 6, 90 and 113.)

[Garcia 2008] Diego Zuquim Guimarães Garcia and Maria Beatriz Felgar de Toledo. *Ontology-Based Security Policies for Supporting the Management of Web Service Business Processes.* In ICSC, 2008. (Cited on page 36.)

[Ghallab 1998] Malik Ghallab, A Howe, C Knoblock, D McDermott, A Ram, M Veloso, Daniel Weld and D Wilkins. *PDDL—The Planning Domain Definition Language.* AIPS98 planning committee, vol. 78, no. 4, page 27, 1998. (Cited on page 32.)

[Giacomo 1995] Giuseppe De Giacomo and Maurizio Lenzerini. *PDL-based framework for reasoning about actions.* In AI*IA, pages 103–114, 1995. (Cited on page 32.)

[Giacomo 2000] Giuseppe De Giacomo, Yves Lespérance and Hector J. Levesque. *ConGolog, a concurrent programming language based on the situation calculus.* Artif. Intell., vol. 121, no. 1-2, pages 109–169, 2000. (Cited on page 33.)

[Giordano 1998] Laura Giordano, Alberto Martelli and Camilla Schwind. *Dealing with Concurrent Actions in Modal Action Logics.* In ECAI, pages 537–541, 1998. (Cited on page 32.)

[Goedertier 2008] Stijn Goedertier. *Declarative Techniques for Modeling and Mining Business Processes.* PhD thesis, Katholieke Universiteit Leuven, Faculteit Economie en Bedrijfswetenschappen, 2008. (Cited on pages 24, 36, 37 and 132.)

[Guermouche 2009a] Nawal Guermouche and Claude Godart. *Asynchronous Timed Web Service-Aware Choreography Analysis.* In CAiSE, 2009. (Cited on page 36.)

[Guermouche 2009b] Nawal Guermouche and Claude Godart. *Timed Model Checking Based Approach for Web Services Analysis.* In ICWS, pages 213–221, 2009. (Cited on pages 5, 24, 36 and 39.)

[Hamadi 2003] Rachid Hamadi and Boualem Benatallah. *A Petri Net-based Model for Web Service Composition.* In ADC, pages 191–200, 2003. (Cited on pages 40 and 44.)

[Hinz 2005] Sebastian Hinz, Karsten Schmidt 0004 and Christian Stahl. *Transforming BPEL to Petri Nets.* In Business Process Management, pages 220–235, 2005. (Cited on page 40.)

[Hoffmann 2001] Jörg Hoffmann. *FF: The Fast-Forward Planning System.* AI Magazine, vol. 22, no. 3, pages 57–62, 2001. (Cited on page 32.)

[Holzmann 2004] Gerard J. Holzmann. The spin model checker - primer and reference manual. Addison-Wesley, 2004. (Cited on pages 39 and 103.)

[Hopcroft 2001] John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation - (2. ed.). Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001. (Cited on page 39.)

[Janssens 2000] Gerrit K. Janssens, Jan Verelst and Bart Weyn. *Techniques for Modeling Workflows and Their Support of Reuse.* In Business Process Management, pages 1–15, 2000. (Cited on pages 30 and 40.)

[Kallel 2009] Slim Kallel, Anis Charfi, Tom Dinkelaker, Mira Mezini and Mohamed Jmaiel. *Specifying and Monitoring Temporal Properties in Web Services Compositions.* In ECOWS, pages 148–157, 2009. (Cited on page 43.)

[Kautz 1992] Henry A. Kautz and Bart Selman. *Planning as Satisfiability.* In ECAI, pages 359–363, 1992. (Cited on page 32.)

[Kazhamiakin 2006] Raman Kazhamiakin, Paritosh K. Pandya and Marco Pistore. *Representation, Verification, and Computation of Timed Properties in Web.* In ICWS, pages 497–504, 2006. (Cited on page 36.)

[Koehler 1997] Jana Koehler, Bernhard Nebel, Jörg Hoffmann and Yannis Dimopoulos. *Extending Planning Graphs to an ADL Subset.* In ECP, pages 273–285, 1997. (Cited on page 32.)

[Kowalski 1986a] Robert A. Kowalski and Marek J. Sergot. *A Logic-based Calculus of Events.* New Generation Comput., vol. 4, no. 1, pages 67–95, 1986. (Cited on pages 6, 17 and 45.)

[Kowalski 1986b] Robert A. Kowalski and Marek J. Sergot. *A Logic-based Calculus of Events.* New Generation Comput., vol. 4, no. 1, pages 67–95, 1986. (Cited on page 32.)

[Levesque 1997] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin and Richard B. Scherl. *GOLOG: A Logic Programming Language for*

Dynamic Domains. J. Log. Program., vol. 31, no. 1-3, pages 59–83, 1997. (Cited on page 33.)

[Levesque 1998] Hector J. Levesque, Fiora Pirri and Raymond Reiter. *Foundations for the Situation Calculus.* Electron. Trans. Artif. Intell., vol. 2, pages 159–178, 1998. (Cited on pages 32 and 33.)

[Liu 1997] Ling Liu and Calton Pu. *ActivityFlow: Towards Incremental Specification and Flexible Coordination of Workflow Activities.* In ER, pages 169–182, 1997. (Cited on page 31.)

[Lu 2007] Ruopeng Lu and Shazia Wasim Sadiq. *A Survey of Comparative Business Process Modeling Approaches.* In BIS, pages 82–94, 2007. (Cited on pages 30, 37 and 46.)

[Mahbub 2004] Khaled Mahbub and George 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, 2004. ACM. (Cited on pages 6, 26, 44, 47 and 134.)

[Martens 2005] Axel Martens. *Analyzing Web Service Based Business Processes.* In FASE, pages 19–33, 2005. (Cited on page 102.)

[Mccarthy 1963] J. Mccarthy. *Situations, actions and causal laws.* 1963. (Cited on pages 32 and 33.)

[McDermott 1996] Drew V. McDermott. *A Heuristic Estimator for Means-Ends Analysis in Planning.* In AIPS, pages 142–149, 1996. (Cited on page 32.)

[McDermott 2002] Drew V. McDermott. *Estimated-Regression Planning for Interactions with Web Services.* In AIPS, pages 204–211, 2002. (Cited on page 35.)

[McIlraith 2002] Sheila A. McIlraith and Tran Cao Son. *Adapting Golog for Composition of Semantic Web Services.* In KR, pages 482–496, 2002. (Cited on pages 33, 37 and 46.)

[Medjahed 2003] Brahim Medjahed, Athman Bouguettaya and Ahmed K. Elmagarmid. *Composing Web services on the Semantic Web.* VLDB J., vol. 12, no. 4, pages 333–351, 2003. (Cited on page 35.)

[Menzel 2009] Michael Menzel, Ivonne Thomas and Christoph Meinel. *Security Requirements Specification in Service-Oriented Business Process Management.* In ARES, pages 41–48, 2009. (Cited on page 36.)

[Merz 1994] M. Merz, D. Moldt, K. Muller and W. Lamersdorf. *Workflow Modeling and Execution with Coloured Petri Nets in COSM*, 1994. (Cited on pages 30 and 40.)

[Morimoto 2008] Shoichi Morimoto. *A Survey of Formal Verification for Business Process Modeling.* In ICCS (2), pages 514–522, 2008. (Cited on pages 38, 40 and 46.)

[Moser 2010] Oliver Moser, Florian Rosenberg and Schahram Dustdar. *Event Driven Monitoring for Service Composition Infrastructures.* In WISE, pages 38–51, 2010. (Cited on pages 26, 43, 44, 45, 47 and 134.)

[Mueller 2006] Erik T. Mueller. Commonsense reasoning. Morgan Kaufmann Publishers Inc., CA, USA, 2006. (Cited on pages 8, 18, 104, 121, 125, 138 and 139.)

[Narayanan 2002a] S. Narayanan and S. A. McIlraith. *Simulation, verification and automated composition of web services.* In WWW, pages 77–88, 2002. (Cited on pages 40 and 98.)

[Narayanan 2002b] Srini Narayanan and Sheila A. McIlraith. *Simulation, verification and automated composition of web services.* In WWW, pages 77–88, 2002. (Cited on page 34.)

[Nau 1999] Dana S. Nau, Yue Cao, Amnon Lotem and Héctor Muñoz-Avila. *SHOP: Simple Hierarchical Ordered Planner.* In IJCAI, pages 968–975, 1999. (Cited on page 34.)

[Neubauer 2008] Thomas Neubauer and Johannes Heurix. *Defining Secure Business Processes with Respect to Multiple Objectives.* In ARES, 2008. (Cited on page 36.)

[Ouaknine 2005] Joël Ouaknine. *Verification of Reactive Systems: Formal Methods and Algorithms. By Klaus Schneider. Springer, Texts in Theoretical Computer Science Series, 2004, ISBN: 3-540-00296-0, pp 600.* Softw. Test., Verif. Reliab., vol. 15, no. 3, pages 202–203, 2005. (Cited on page 103.)

[Pednault 1994] Edwin P. D. Pednault. *ADL and the State-Transition Model of Action.* J. Log. Comput., vol. 4, no. 5, pages 467–512, 1994. (Cited on page 32.)

[Peer 2005] Joachim Peer. *Web Service Composition as AI Planning Ð a Survey.* Language, no. March, 2005. (Cited on page 32.)

[Penberthy 1992] J. Scott Penberthy and Daniel S. Weld. *UCPOP: A Sound, Complete, Partial Order Planner for ADL.* In KR, pages 103–114, 1992. (Cited on page 32.)

[Pesic 2006] Maja Pesic and Wil M. P. van der Aalst. *A Declarative Approach for Flexible Business Processes Management.* In Business Process Management Workshops, 2006. (Cited on pages 25, 37, 46 and 133.)

[Pirri 1999] Fiora Pirri and Raymond Reiter. *Some Contributions to the Metatheory of the Situation Calculus*. J. ACM, vol. 46, no. 3, pages 325–361, 1999. (Cited on pages 32 and 33.)

[Ponge 2007] Julien Ponge, Boualem Benatallah, Fabio Casati and Farouk Toumani. *Fine-Grained Compatibility and Replaceability Analysis of Timed Web Service Protocols*. In ER, 2007. (Cited on page 36.)

[Ponnekanti 2002] Shankar R. Ponnekanti and Armando Fox. *SWORD: A developer toolkit for web service composition*. In Proceedings of the 11th International WWW Conference (WWW2002), Honolulu, HI, USA, 2002. (Cited on page 35.)

[Quartel 2004] Dick A. C. Quartel, Remco M. Dijkman and Marten van Sinderen. *Methodological support for service-oriented design with ISDL*. In ICSOC, pages 1–10, 2004. (Cited on page 36.)

[Rao 2003] Jinghai Rao, Peep Küngas and Mihhail Matskin. *Application of Linear Logic to Web Service Composition*. In ICWS, pages 3–, 2003. (Cited on page 35.)

[Rao 2004a] J. Rao and Xiaomeng Su. *A Survey of Automated Web Service Composition Methods*. In SWSWPC, 2004. (Cited on pages 30, 31 and 46.)

[Rao 2004b] Jinghai Rao, Peep Küngas and Mihhail Matskin. *Logic-based Web Services Composition: From Service Description to Process Model*. In ICWS, pages 446–453, 2004. (Cited on page 35.)

[Redavid 2008] D. Redavid, L. Iannone, T. R. Payne and G. Semeraro. *OWL-S Atomic Services Composition with SWRL Rules*. In ISMIS, pages 605–611, 2008. (Cited on page 98.)

[Reichert 1998] Manfred Reichert and Peter Dadam. *ADEPT$_{flex}$-Supporting Dynamic Changes of Workflows Without Losing Control*. J. Intell. Inf. Syst., vol. 10, no. 2, pages 93–129, 1998. (Cited on page 31.)

[Rodríguez 2007] Alfonso Rodríguez, Eduardo Fernández-Medina and Mario Piattini. *A BPMN Extension for the Modeling of Security Requirements in Business Processes*. IEICE Transactions, vol. 90-D, 2007. (Cited on page 36.)

[Röglinger 2009] Maximilian Röglinger. *Verification of Web Service Compositions: An Operationalization of Correctness and a Requirements Framework for Service-oriented Modeling Techniques*. Business & Information Systems Engineering, vol. 1, no. 6, pages 429–437, 2009. (Cited on pages 46, 102 and 103.)

[Rozinat 2008] Anne Rozinat and Wil M. P. van der Aalst. *Conformance checking of processes based on monitoring real behavior*. Inf. Syst., vol. 33, no. 1, pages 64–95, 2008. (Cited on page 102.)

[Russell 1995] Stuart J. Russell and Peter Norvig. Artificial intelligence - a modern approach: the intelligent agent book. Prentice Hall series in artificial intelligence. Prentice Hall, 1995. (Cited on page 32.)

[Russell 2006] Nick Russell, Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. *Workflow Exception Patterns.* In CAiSE, pages 288–302, 2006. (Cited on pages 43 and 44.)

[Sacerdoti 1974] Earl D. Sacerdoti. *Planning in a Hierarchy of Abstraction Spaces.* Artif. Intell., vol. 5, no. 2, pages 115–135, 1974. (Cited on page 34.)

[Sadiq 1999] S. W. Sadiq and M. E. Orlowska. *On capturing process requirements of workflow based business information systems.* In BIS'99, 1999. (Cited on page 31.)

[Salaün 2004] Gwen Salaün, Lucas Bordeaux and Marco Schaerf. *Describing and Reasoning on Web Services using Process Algebra.* In ICWS, pages 43–, 2004. (Cited on page 41.)

[Schlingloff 2005] Bernd-Holger Schlingloff, Axel Martens and Karsten Schmidt 0004. *Modeling and Model Checking Web Services.* Electr. Notes Theor. Comput. Sci., vol. 126, pages 3–26, 2005. (Cited on page 102.)

[Schuster 2000] Hans Schuster, Dimitrios Georgakopoulos, Andrzej Cichocki and Donald Baker. *Modeling and Composing Service-Based and Reference Process-Based Multi-enterprise Processes.* In Proceedings of the 12th International Conference on Advanced Information Systems Engineering, CAiSE '00, pages 247–263, London, UK, 2000. Springer-Verlag. (Cited on page 31.)

[Shanahan 2000] Murray Shanahan. *An abductive event calculus planner.* J. Log. Program., vol. 44, no. 1-3, pages 207–240, 2000. (Cited on page 32.)

[Sirin 2002] Evren Sirin, James Hendler and Bijan Parsia. *Semi-automatic Composition of Web Services using Semantic Descriptions.* In In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003, pages 17–24, 2002. (Cited on page 35.)

[Sirin 2004] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler and Dana S. Nau. *HTN planning for Web Service composition using SHOP2.* J. Web Sem., vol. 1, no. 4, pages 377–396, 2004. (Cited on pages 34, 37 and 46.)

[Song 2006] Haitao Song, Yanming Sun, Yingyu Yin and Shixiong Zheng. *Dynamic Weaving of Security Aspects in Service Composition.* In SOSE, 2006. (Cited on page 36.)

[Souza 2009] Andre R. R. Souza, Bruno L. B. Silva, Fernando Antônio Aires Lins, Julio C. Damasceno, Nelson Souto Rosa, Paulo R. M. Maciel, Robson W. A. Medeiros, Bryan Stephenson, Hamid R. Motahari Nezhad, Jun Li and Caio

Northfleet. *Incorporating Security Requirements into Service Composition: From Modelling to Execution.* In ICSOC/ServiceWave, 2009. (Cited on pages 36 and 86.)

[Subrahmanian 1995] V. S. Subrahmanian and Carlo Zaniolo. *Relating Stable Models and AI Planning Domains.* In ICLP, pages 233–247, 1995. (Cited on page 32.)

[Sun 2009] Mingjie Sun, Bixin Li and Pengcheng Zhang. *Monitoring BPEL-Based Web Service Composition Using AOP.* In ACIS-ICIS, pages 1172–1177, 2009. (Cited on page 43.)

[Suntinger 2008] Martin Suntinger, Hannes Obweger, Josef Schiefer and M. Eduard Gröller. *The Event Tunnel: Interactive Visualization of Complex Event Streams for Business Process Pattern Analysis.* In PacificVis, pages 111–118, 2008. (Cited on page 43.)

[Sven 2002] Lämmermann Sven. *Runtime Service Composition via Logic-Based Program Synthesis.* PhD thesis, KTH, Microelectronics and Information Technology, IMIT, 2002. (Cited on page 35.)

[Tate 1977] Austin Tate. *Generating Project Networks.* In IJCAI, pages 888–893, 1977. (Cited on page 32.)

[Tondello 2008] G. F. Tondello and Frank Siqueira. *The QoS-MO ontology for semantic QoS modeling.* In SAC, pages 2336–2340, 2008. (Cited on page 97.)

[Tsamardinos 2003] Ioannis Tsamardinos, Thierry Vidal and Martha E. Pollack. *CTP: A New Constraint-Based Formalism for Conditional, Temporal Planning.* Constraints, vol. 8, no. 4, pages 365–388, 2003. (Cited on page 36.)

[van der Aalst 2006] Wil M. P. van der Aalst and Maja Pesic. *DecSerFlow: Towards a Truly Declarative Service Flow Language.* In The Role of Business Processes in Service Oriented Architectures, 2006. (Cited on pages 5 and 101.)

[Vanhatalo 2007] Jussi Vanhatalo, Hagen Völzer and Frank Leymann. *Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition.* In ICSOC, pages 43–55, 2007. (Cited on pages 43 and 44.)

[Vanhatalo 2008] Jussi Vanhatalo, Hagen Völzer, Frank Leymann and Simon Moser. *Automatic Workflow Graph Refactoring and Completion.* In ICSOC, pages 100–115, 2008. (Cited on pages 43 and 44.)

[Verma 2005] Kunal Verma and Amit P. Sheth. *Autonomic Web Processes.* In ICSOC, pages 1–11, 2005. (Cited on page 44.)

[Waldinger 2000] Richard J. Waldinger. *Web Agents Cooperating Deductively.* In FAABS, pages 250–262, 2000. (Cited on page 35.)

[Weske 2007] Mathias Weske. Business process management: Concepts, languages, architectures. Springer, 2007. (Cited on page 43.)

[Wikarski 1996] Dietmar Wikarski. *An Introduction to Modular Process Nets.* TR-96-019, International Computer Science Institute, Berkeley, CA, pages 1–51, April 1996. (Cited on pages 30 and 40.)

[Wu 2008] Guoquan Wu, Jun Wei and Tao Huang. *Flexible Pattern Monitoring for WS-BPEL through Stateful Aspect Extension.* In ICWS, 2008. (Cited on page 43.)

[Yi 2004] Xiaochuan Yi and Krys Kochut. *A CP-nets-based Design and Verification Framework for Web Services Composition.* In ICWS, pages 756–760, 2004. (Cited on page 40.)

[Younes 2003] Håkan L. S. Younes and Reid G. Simmons. *VHPOP: Versatile Heuristic Partial Order Planner.* J. Artif. Intell. Res. (JAIR), vol. 20, pages 405–430, 2003. (Cited on page 32.)

[Zahoor 2009a] Ehtesham Zahoor, Olivier Perrin and Claude Godart. *An Integrated Declarative Approach to Web Services Composition and Monitoring.* In WISE, pages 247–260, 2009. (Not cited.)

[Zahoor 2009b] Ehtesham Zahoor, Olivier Perrin and Claude Godart. *Rule-Based Semi Automatic Web Services Composition.* In SERVICES I, pages 805–812, 2009. (Cited on pages 60, 96 and 122.)

[Zahoor 2010a] Ehtesham Zahoor, Olivier Perrin and Claude Godart. *DISC: A declarative framework for self-healing Web services composition.* In ICWS, 2010. (Cited on pages 8, 26, 45, 48, 101, 125, 134 and 138.)

[Zahoor 2010b] Ehtesham Zahoor, Olivier Perrin and Claude Godart. *DISC-SeT: Handling Temporal and Security Aspects in the Web Services Composition.* In ECOWS, 2010. (Cited on pages 8, 125 and 138.)

[Zahoor 2011] Ehtesham Zahoor, Olivier Perrin and Claude Godart. *An Event-Based Reasoning Approach to Web Services Monitoring.* In ICWS, pages 628–635, 2011. (Cited on pages 111 and 117.)

[Zhang 2004] Jia Zhang, Jen-Yao Chung, Carl K. Chang and Seongwoon Kim. *WS-Net: A Petri-net Based Specification Model for Web Services.* In ICWS, pages 420–427, 2004. (Cited on page 40.)