# Synthesizing and Runtime Monitoring of Business Process Workflows

THESIS

Submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY

by

Nihita Goel

Under the supervision of
Prof R.K. Shyamasundar

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA
2012

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI (RAJASTHAN)**

# CERTIFICATE

This is to certify that the thesis entitled **Synthesizing and Runtime Monitoring of Business Process Workflows** which is submitted for award of Ph.D. Degree of the Institute, embodies original work done by **Nihita Goel** (ID: 2005PHXF435P) under my supervision.

Signature in full of Supervisor:

Name of Supervisor: R. K. SHYAMASUNDAR

Designation of Supervisor: Senior Professor, Tata Institute of Fundamental Research

Date:

*To my husband and son*

# Acknowledgements

I am deeply indebted to my adviser, Prof. R. K. Shyamasundar, for his support and guidance during the course of this thesis. The work in this thesis has benefitted a great deal from his insights and clarity of thought. I would like to thank him for giving me this opportunity and all the support and privileges he provided me during my Ph.D. His insights, support and encouragement enabled me to do more than what i had imagined.

I would like to thank all the members of the Information Systems Development Group (ISDG) and the School of Technology and Computer Science (STCS), TIFR, for the help they extended during various stages of my work. In particular, I thank Prof. Jaikumar Radhakrishnan, Chairperson ISDG Committee for permitting me to pursue my studies along with my job responsibilities. I thank Dr. N. Raja for helping me with my queries and in reviewing my draft thesis. I thank Dr. N. V. Narendra Kumar for being a collaborator in a part of the work related to the thesis and the members of ISDG and STCS for their cooperation and help especially Sarita, Sandeep, Sushma, John Barretto and Pravin Bhuwad. I would also like to thank my seniors, friends and colleagues from TIFR, Prof. Vahia, Mr. Abhyankar and Mr. Nandagopal for their moral support and encouragement during this period.

I thank BITS Pilani for allowing me to carry out my work. I thank Prof Navneet Goyal, Head, Department of Computer Science, BITS Pilani for his help and support and the other members of the DAC for their valuable inputs during the review of this thesis.

I thank my parents, my in-laws, my sister and all my friends and colleagues for their moral support during the course of my work.

And finally, I express my heartfelt thanks to my husband and son for their patience and encouragement all these long years. This thesis would not have been possible without their support.

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| B2B | Business To Business |
| BLA | Business Level Agreement |
| BLO | Business Level Objective |
| BNF | Backus-Naur Form |
| BPD | BPMN Core Diagram |
| BPEL | Business Process Execution Language |
| BPM | Business Process Management |
| BPMN | Business Process Modeling Notation |
| DSL | Deterministic SL |
| ERP | Enterprize Resource Planning |
| GEF | Graphical Editing Framework |
| M2M | Model To Model |
| M2T | Model To Text |
| MDA | Model Driven Architecture |
| QoS | Quality of Service |
| SLA | Service Level Agreement |
| SLO | Service Level Objective |
| SOA | Service Oriented Architecture |
| SOAP | Simple Object Access Protocol |
| UDDI | Universal Description Discovery Integration |

| | |
|---|---|
| UML | Unified Modeling Language |
| WCP | Workflow Control-flow Patterns |
| WFMC | Workflow Management Coalition |
| WFMS | Workflow Management System |
| WSDL | Web Service Description Language |
| WSFL | Web Service Flow Language |
| XPDL | XML Process Definition Language |
| YAWL | Yet Another Workflow Language |

# List of Symbols

| | |
|---|---|
| $\psi$ | SL formulas also called safety formulas |
| $\varphi$ | Past formulas |
| $\bullet$ | the classical "previous" operator of temporal logic with past |
| $\neg \bullet \neg$ | the dual operator of "previous" operator |
| $\exists$ | existential quantifier |
| $\Box$ | the classical "always" operator |
| $O$ | a set of observables |
| $\tau$ | a trace on a set of observables |
| $\omega$ | an element of trace $\tau$ used in definitions associated with SL/DSL formula |
| $\wedge$ | the "and" operator used in a SL/DSL formula |
| $\vee$ | the or operator used in a SL/DSL formula |
| $\supset$ | the "implies" operator as used in a SL/DSL formula |
| $\equiv$ | the "equivalence" operator as used in a SL/DSL formula |
| A($\psi$) | automata associated with a SL/DSL formula $\psi$ |
| A($\varphi$) | automata associated with a past formula $\varphi$ |
| $\lambda$ | a ternary relation $Q \times O \times Q$ |
| $Prop$ | a set of propositional symbols used in a SL/DSL formula |
| $Aux$ | a set of auxiliary symbols used in a SL/DSL formula |
| $T$ | a transition function for a Mealy machine / set of tasks in a given BPMN Diagram |
| $G$ | output function for a Mealy machine |
| $G_{ps}$ | parallel fork gateways in a BPD Diagram |
| $G_{pm}$ | parallel join gateways in a BPD Diagram |
| $G_{es}$ | exclusive-or gateway in a BPD Diagram |
| $G_{em}$ | exclusive-or merge gateway in a BPD Diagram |
| $G_{ds}$ | event-based gateway in a BPD Diagram |
| $G_{is}$ | inclusive-or gateway in a BPD Diagram |
| $G_{im}$ | inclusive-or merge gateway in a BPD Diagram |
| $G_{cs}$ | complex gateway in a BPD Diagram |
| $G_{cm}$ | complex merge gateway in a BPD Diagram |
| $G_{par}$ | parallel combinator of Orc used in a OrcDAG |
| $G_{where}$ | pruning combinator of Orc used in a OrcDAG |
| $G_{sync}$ | a synchronization operator of Orc used in a OrcDAG |
| $G_{xor}$ | exclusive-or split gateway of Orc used in a OrcDAG |
| $G_{merge}$ | merge combinators of Orc used in a OrcDAG |
| $\triangleq$ | definition |

# Abstract

Organizations today are increasingly automating their business processes using the workflow technologies. These technologies are continuously evolving with the advances in the field of web services, cloud computing and distributed computing. The business process workflows are primarily composed of internal or external web services and are based on Service Oriented Architectures (SOA). Such process workflows require a very systematic approach towards development, starting from a "conceptual" or an abstract model designed by their business analysts, to an "executable" model provided by the technical experts. MDA (Model Driven Architecture) is an OMG initiative that attempts to bridge the gap between the business functionality and the implementation models through the application of model transformations. The thesis aims to apply formal methodologies to develop efficient, correct and inter-operable workflow systems.

The thesis first proposes a framework for the synthesis of business workflows using a "model driven" approach wherein the process workflow is expressed as a conceptual model using BPMN (Business Process Modeling and Notation) which is the de facto notation for conceptual modeling of such systems and is widely used in Model Drive Architectures for enterprize-scale solutions. From a conceptual model in BPMN, the executable model is derived in Orc (a recent language for web orchestrations) through transformations. The transformation system, called Bpmn2Orc, realizes an executional platform for business process workflows using Orc, and hence, extends the capabilities of the language Orc to the business analysts community and the cloud computing environments also. The language Orc, developed at the University of Texas, is based on a few basic constructs and provides a clean separation between computation and orchestration and has been used to model different kinds of orchestrations and workflow patterns. To achieve the above mentioned transformation, the languages BPMN and Orc are first analysed, and each BPMN core element is mapped to an equivalent Orc construct. Certain graph based transformation techniques are then applied where-in a Business Process Diagram (BPD) is validated and then converted to a set of Orc computation structures (in the form of Orc Graphs). These Orc Graphs are then used to perform validations of the conceptualized workflow in view of the executional semantics. Later, it is traversed to generate the executable code that can be used by the technical analysts for realizing the process workflow. These transformations are described along with an implementation and illustrated with an example. Thus, the system Bpmn2Orc provides a validated code generation in Orc starting from a BPMN specification.

In order to arrive at the choice of executable language Orc, an extensive study and analysis of various languages supporting business process workflows is undertaken. Orc is analysed for its support of forty three workflow patterns and compared with the equivalent pattern support in other languages. The results of this study including the details of each pattern support in Orc and an in-depth comparison with other languages is also included as part of this work.

The process workflows are often composed of "external" web services. Their execution involves task executions that are outside their control domain. Thus, one of the biggest challenge in realizing such complex workflows is to ensure that their behavior remains consistent with the intended specifications (note that our transformation validates only the BPMN specification and not the underlying concepts of the model to meet the performance requirements that range from QoS to real-time constraints). Thus, *runtime* monitoring is required as it is difficult to perceive all possible outcomes of a component service at the design time and to evaluate dynamic executions. Further, a workflow may require certain values, for example, QoS attributes, that can only be computed externally. Also, workflows that use service "discoveries" need certain values at "runtime" to query their registry databases and find the "best" fitting service. Very often, the organizations also enter into a Service Level Agreement (SLA) with their partners in order to guarantee their process behavior. An online monitoring system named WF_SLA_MON, is developed as part of this work, and is applied for such distributed workflows. It works by maintaining a watch on all external interactions in order to ensure that a set of given properties are always satisfied. The system executes concurrently with the runtime system of the process workflow and enables the workflow to initiate action on detection of any property violation. Properties may be functional or non functional. The non-functional properties are specified using user or system provided macros. The functional properties may be specified either

1. in SL, a class of temporal logic that has the expressive power of regular safety properties or

2. as wanted/unwanted scenarios using Message Sequence Charts(MSC).

The system identifies properties with "basic" or "composite" metrics. Basic metrics can be aggregated to form composite metrics. From the given properties, the observers are derived as a deterministic finite state automata and integrated with the underlying workflow engine such that the property violations can not only be detected but also acted upon on a real time basis. The application of the system is illustrated through examples of monitoring properties of a BPMN system. To summarise, the WF_SLA_MON system enables the workflow service provider to run-time monitor a given BPMN system with respect to properties related to process interactions with external stake holders and then adapt itself as per specification.

As part of the work, the monitoring system is applied for a complex SLA. The SLA is first broken into computable and verifiable properties and integrated with the process workflow in such a way that it can indeed be ensured. The complete process is illustrated in the thesis.

# Contents

# Chapter 1

# Introduction

Today's business environment is characterized by increasing levels of competition. Recent advances in the field of Information Technology (IT) enable enterprises to increase their market share and make profits by adapting (or re-engineering) their business processes to these advances. They can make their tasks easier and efficient by redesigning their organization and changing the way they work to obtain spectacular improvement. Amongst the key IT enablers are web services, service oriented architectures and the very recent cloud computing environments. Together, they provide enterprises with process automation tools and techniques and the associated computing infrastructure in an unprecedented manner.

A business process of any organization is a collection of "activities" or "tasks" that are performed by an individual or an automated system in order to produce a specific service or product required by the organization. These are often referred as business process workflows. Traditional workflow technology which is here since mid 1990's was concerned about simple task flow and monitoring in a single enterprise domain. However, today's business processes run across organizational boundaries and newer models of business, leveraging the service based architectures are now emerging. This thesis, provides a framework for synthesizing and runtime monitoring cross organizational workflows. Formal methodologies are used to develop an efficient, correct and inter-operable workflow system. To explain this and to set the context, it is important to first provide a historical perspective for business computing and the other necessary background of web services and service oriented architectures. Later, the main research challenges are discussed followed by the objective and motivation of the work.

## 1.1 Historical perspective

In order to gain insight about workflow management systems, it is useful to consider the evolution of business computing over the last few decades. Figure 1.1 provides the business computing in a historical perspective as described by Aalst et al [van der Aalst, 1998]. The figure describes the architecture of a typical information system used for business computing in terms of its components. The evolution shows that more and more generic tasks have been taken out of programs and put into separate management systems. The evolution is outlined here. The denoted time periods are not strict but more to depict the progressive development.

*1965-1975: Stand-alone Applications:* Early information systems were composed of many stand-alone applications. For each of these applications an application-specific user interface and database system had to be developed. Each application had its own interface or libraries for user interaction as well as data storage and retrieval. The applications therefore had data redundancy. The most common programming language used that time was COBOL. It did not support local variables, recursion, dynamic memory allocation, or structured programming constructs earlier. (Support for some or all of these features has been added in later editions of COBOL). COBOL programs were and are still used in military and government agencies and even commercial organizations under the operating systems like IBM z/OS, the POSIX family, as well as Microsoft Windows.

*1975-1985: Database Management Systems:* In the seventies, data was pushed out of applications and managed separately. For this purpose database management systems (DBMSs) were developed. The early DBMS systems were mainly relational, for example, INGRES, IBMs PRTV etc. By using these systems, applications were freed from the burden of data management. Data redundancy could also be reduced from these applications.

*1985-1995: User Interface Management:* In the eighties a similar thing happened for user interfaces and User Interface Management Systems (UIMSs) emerged which enabled application developers to push the user interaction out of the applications. The basic concepts of an UIMS were articulated at a workshop on Graphical Input and Interactions Techniques in 1982 [Löwgren, 1988]. The emphasis was on models and the first model in this direction was the "Seeheim" model. Since then there has been continuous development in this field with each system requiring a particular style of programming. These systems helped the developers by providing a library using which user interfaces could be defined rapidly and the designer could do this in a standard way. Some examples of UIMS are Macintosh Toolbox, MacApp, COUSIN etc. Graphical based user interfaces quickly replaced the character ones, and as a result the utilization of UIMS also increased. Today the functions of UIMS are integrated in other tools like DBMS packages, programming interfaces and web browsers. Around the same time the DBMS technologies also advanced to include Structured Query Language (SQL) and evolved into more sophisticated products like Sybase, Informix etc.

*1995-2005: Workflow Management:* Workflow management systems saw their entry in mid nineties pushing generic business procedures out of the applications. One of the early workflow system is the FileNet's business process automation software "WorkFlo". Earlier workflow systems were used mainly in the manufacturing industry but the scope grew slowly into business process re-engineering and office automation products also. Workflow systems provided a mechanism to control the flow of information and work within an en-

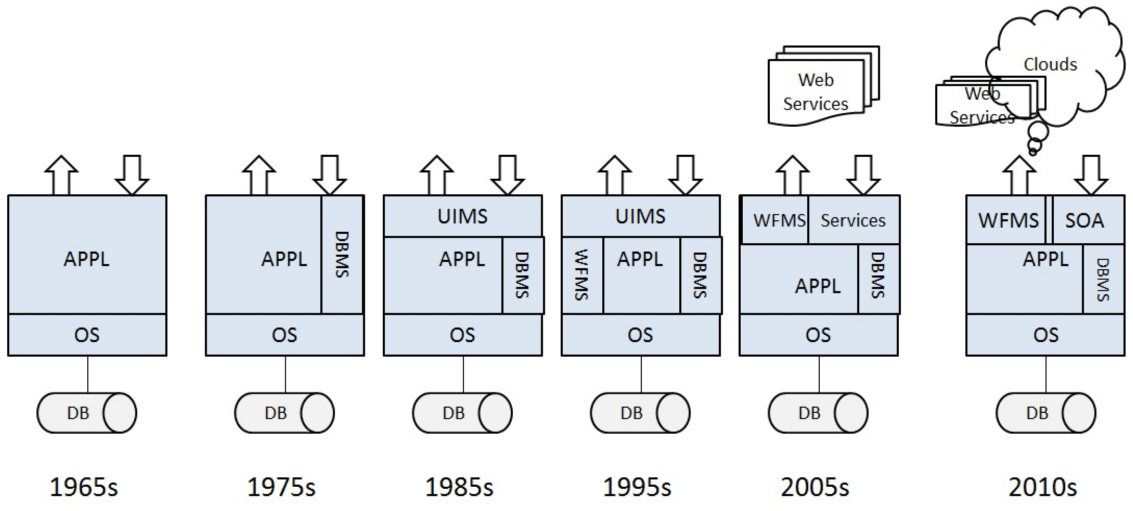


Figure 1.1: Business computing historical perspective. Source [van der Aalst, 1998]

terprise.

*2005-2010: Service Oriented Architectures and Clouds:* Over the last decade the Service Oriented Architectures (SOAs) have emerged and have found their way in WFMS which allows inter organizational business process workflows to use internet based web services. The very recent emergence of cloud computing infrastructure is likely to change the face of WFMS further by allowing organizations to share and compose business processes including data over the cloud.

To conclude with the historical perspective in workflow management, some early work in this field is also worth mentioning. The idea to have such generic tools for supporting business processes actually emerged in 1970s with pioneers such as Skip Ellis and Michael Zisman. Zisman completed his Ph.D. thesis “Representation, Specification and Automation of Office Procedures” in 1997. Ellis and others on the other hand worked on “Office Automation Systems” in 1990s where they used petri net based workflow models [van der Aalst and vanHee, 2004]. There are several reasons as to why it took such a long time before workflow management systems became established as a standard component even after some of these early works. First, it requires users to be linked to a computer network. Only in late 1990s with the emergence of internet did people also become easily connected to the network. Second, earlier information systems evolved from local systems that were unaware of overall business processes which led to automation in parts and therefore workflow could not be considered as any new piece of functionality. Finally, the rigid and inflexible character of the early products, procedures and systems scared away many potential users.

In the next section we will briefly discuss about the internet and web-services which were the catalysts for the evolution of workflow systems today.

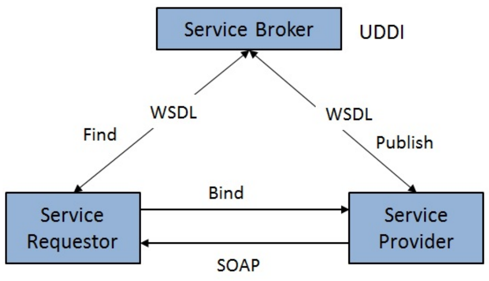


Figure 1.2: Find-Bind-Publish paradigm

## 1.2 Impact of Internet and the Web Services

The W3C defines a "Web service" as *a software system designed to support inter-operable machine-to-machine interaction over a network*. Web Services use the find-bind-publish paradigm as shown in Figure 1.2. In this paradigm, service providers register their service in a public Universal Description Discovery and Integration (UDDI) registry. This registry is used by consumers to find services that match certain criteria. UDDI defines a set of Simple Object Access Protocol (SOAP) messages that provide the client API for accessing a registry. If the registry has such a service, it provides the consumer with a contract and an endpoint address for that service. Using this, the service consumer binds to the service and then invokes it.

While the emergence of the internet provided the connectivity between people across organization and even crossing the boundaries of nations, web services provided a mechanism for the organizations to integrate their business processes. The emergence of these two technologies was therefore considered as a significant step in the evolution of workflows, especially, the Business to Business (B2B) collaborations. Using these web services one could move the "activities" or the "tasks" of a workflow across the organizational domain. It also served as an interface for the process to connect with legacy applications and to move towards service oriented architectures which we will describe in the next section.

The web service stack is given in Figure 1.3. Here, Web Services Description Language, known by the acronym WSDL, is an interface described in a machine-processable format, using which, other systems interact with the given web service in a manner prescribed by its description. Interaction is through messages, generally SOAP messages. SOAP Messages are typically conveyed using HTTP or other web related standards with an XML serialization. On top of the WS-Stack, there are two elements to be found: Orchestration and Choreography, which provides a mechanism for composition of the web services.

A business process is often formed by combining web services (process fragments) in various ways to deliver new or modified business capability. Such compositions can be seen to have both an internal and external view. The internal view is seen as "orchestration" and defines the actual or intended internal behavior of the process fragment. It includes

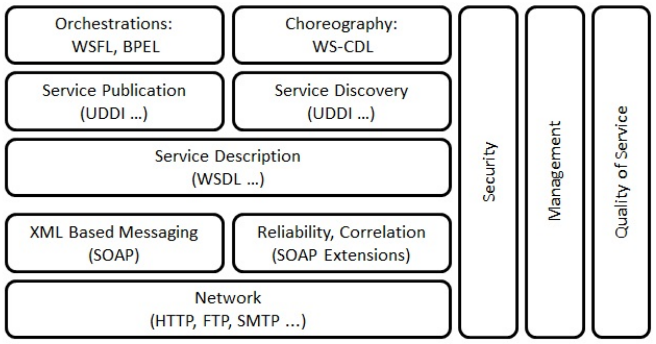


Figure 1.3: Web-service Stack

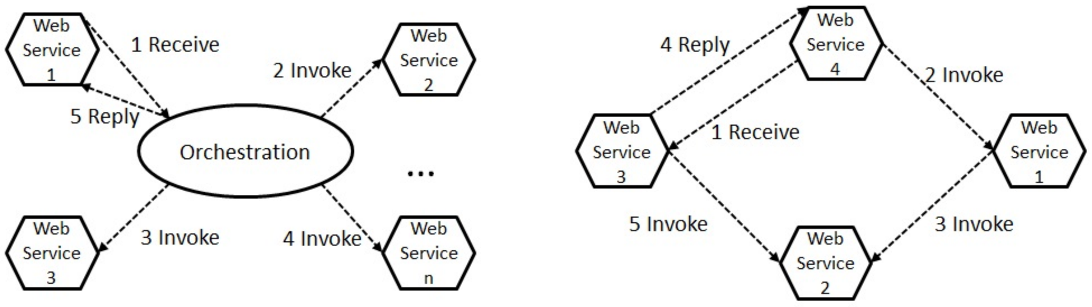


Figure 1.4: Orchestration v/s Choreography in Web services

the activities, transitions and the internal resources required to support the enactment of the process. The external view or the "choreography" defines the behavior of the fragment as a black box, seen from the outside and addressed through its interfaces. This view sees the process fragment very much as a source and sink of messages or events of different types. Choreography v/s Orchestration is depicted in Figure 1.4.

## 1.3 Emergence of Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is an architectural style that emphasizes implementation of components as services. It promotes loose coupling between software components so that they can be reused. Applications in SOA are built based on "services" where the service is an implementation of a well-defined business functionality, and such services can

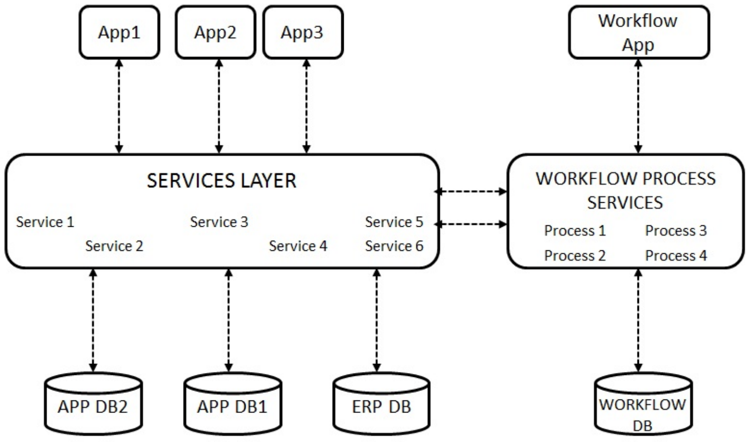


Figure 1.5: SOA architecture illustrated

then be consumed by clients in different applications or business processes. An important aspect of SOA is the separation of the service interface (the what) from its implementation (the how). Web Services are the key enabler of SOA. They are consumed by clients that are not concerned with how these services will execute their requests. Services can be dynamically discovered and composite services can be built from aggregates of other services

Figure 1.5 shows how various services lie under the service layer as components. Applications access services via this layer. The services can be external or internal. In case of external services the service layer of the external application (organization) would be used. Workflow applications also use the functionality of these applications via these services.

The service-oriented computing (SOC) paradigm uses services to support the development of rapid, low-cost, inter-operable, and distributed applications. "Services" in SOA are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled in novel ways. They perform functions that range from simple mathematical calculations or execute complex business processes running across different service consumers and providers. The approach is independent of specific programming languages or operating systems. It lets organizations expose their core competencies programmatically over the Internet using standard XML-based languages and protocols and a self-describing interface.

## 1.4 Workflow Management Systems

Having looked at the basics of the SOA technologies, we will now delve into Workflow systems as such. The Workflow Management Coalition (WFMC), a non-profit international organization of workflow vendors, users, analysts and university/research groups defines a workflow as: *the automation of a business process, in whole or in part, during which documents, information, or tasks are passed from one participant to another, according to a set of procedural rules* [Stohr and Zhao, 2001]. The goal of workflow is the automation of complete processes instead of isolated tasks. Workflows based on SOA promotes inter-organizational collaborations by integrating the teams involved in managing different parts of a workflow.

A Workflow Model / Specification is used to define a workflow both at the task and structure levels. There are two types of workflows, namely "Abstract" and "Concrete". While concrete workflows are also referred to as executable workflows, an abstract model, includes tasks that are described in an abstract form without referring to specific resources for task execution. Abstract or conceptual models provide the ability to define workflows in a flexible way, isolating execution details. Furthermore, an abstract model provides only service end point information on how the workflow has been composed and therefore the sharing of workflow descriptions between users is feasible. In the concrete model, the tasks of the workflow are bound to specific resources and therefore this model provides service execution information on how the workflow has been composed (e.g. dataflow bindings, control flow structures).

A workflow can be executed by a Workflow Management System (WFMS). WFMC define a WFMS as: *a system that defines, creates, and manages the execution of workflows through the use of software running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invokes the use of IT-tools and applications.* These systems thus support the execution of business processes in an organization. They take care of the distribution of work (and associated data) to the right people at the right time. They are configured based on (often graphical) process models. Similar to database management systems, WFMSs are also generic systems, but instead of supporting the storage and retrieval of data, they focus on the distribution of work to resources and provide facilities for monitoring its progress and managing it to completion.

It may be worth mentioning here that while business workflows are applicable for business domains, scientific workflows often underlies many large-scale complex e-science applications such as climate modeling, structural biology and chemistry, medical surgery or disaster recovery simulation. Compared with business workflows, scientific workflows have special features such as computation, data or transaction intensity, less human interaction, and a large number of activities. Some emerging computing infrastructures such as grid and cloud computing have powerful computing and resource sharing capabilities through which these special features can be accommodated. One of the major differences between a business and scientific workflow is that while a business workflow is "control flow" centric a scientific workflow is "data/information flow centric". This is the primary reason why the business workflow models cannot be used directly for the scientific workflows. The suitability of these workflow models under scientific domain is currently an active area of research.

Most of the contemporary WFM systems are built on a Service Oriented Architecture

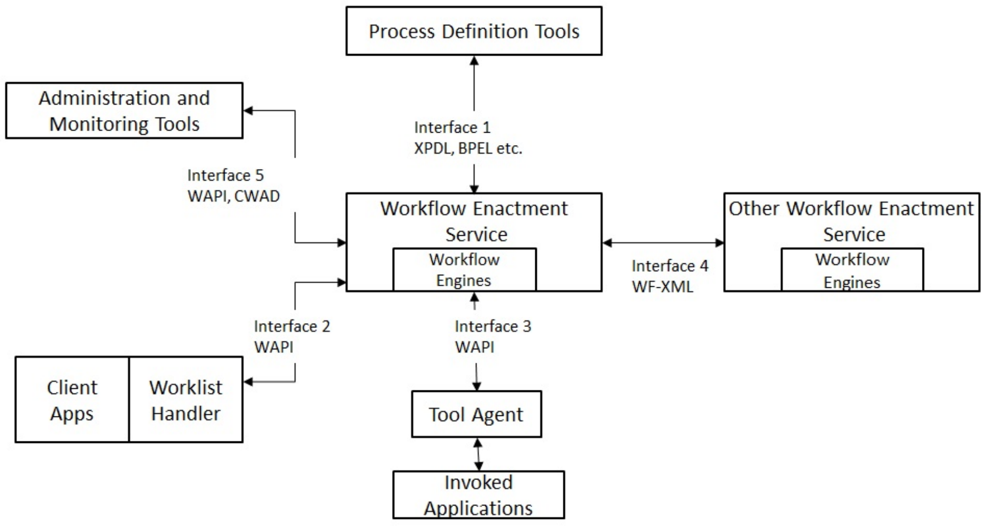


Figure 1.6: A workflow reference model

where the application functionality is not provided by one large monolithic application (as in the traditional model), but by services that are offered by providers. Composite services are composed out of other services, that can also be provided as services. To facilitate the creation of composite services, mechanisms for monitoring, conformity checking and coordinating the different sub-services are needed. In addition to this is the Services-Management, providing services like assurance, certification and rating of services. With SOA, systems developed on different platforms and technologies, such as, legacy systems, Java, and .Net applications, are able to communicate directly using standardized interfaces and protocols. The definition and execution of business processes is achieved using a common workflow or a web service orchestration language e.g. BPEL, XPDL, WSFL etc. BPEL is currently the most widely accepted industrial standard and is a hybrid between IBMs graph-based WSFL and Microsoft's block based XLANG which have their origin in Petri nets and pi calculus respectively. Other languages like YAWL and Orc are still in the domain of research but have been found very promising to support the complexities involved.

The WFMC reference model for a workflow system given in mid 1990's is given in Figure 1.6. Central to the reference model is the workflow enactment service, that consists of one or more workflow engines. These engines execute the workflows by initiating new processes, assigning task to users or roles, listening to events etc. The *first* interface concerns the specification of the process or the process definition tools. The processes that are executed can be defined at design time by means of a graphical process editor. A process definition language should be based on formal notation, for example, Petri-nets, as then these specifications can be verified and validated. BPMN and XPDL (the XML Process

Definition Language) are some standard languages for defining and interchanging processes designed by the WFMS. The *second* interface is with the workflow client applications which is usually implemented by a graphical interface that consists of a *worklist* with the work items that have to be performed by the logged in person. The workflow engine does the assignment of tasks to the employees (resources). This can be done by means of a push, or a pull system. Most WFMS work with a pull system, because a push-system has the disadvantage that tasks can be pushed into the worklist of an employee that is unavailable for some reason. The WAPI (Workflow Application Program Interface) is normally used as an interface here. The *third* interface allows the WFMS to interact with external applications, either by linking with external applications like Document Management Systems, JDBC or invoking external web services. The *fourth* interface allows different WFMS to communicate with each other, i.e. that at a certain point in the process, the WFMS can start a (sub-) process in another WFMS. WF-XML is normally used as a standard here. The *fifth* interface is for administrative and monitoring tools. These tools can be used to control issues like status, progress, and workload, and to add new users and roles.

Web orchestration languages like BPEL are often extended e.g. BPEL extensions, to allow for the additional requirements of WFM, mainly, manual tasks handled by humans. Such extensions involve two parts of the WFM system. First, the language is extended with a new task type, e.g. a Human Activity, or a Staff type, which allows the modeling of workflows containing tasks to be handled by humans. Second, the WFM system must contain some kind of task portal, where process participants can claim and executetasks assigned to them.

## 1.5 Business Process Management (BPM)

A parallel terminology for the above described workflow systems is now emerging as (Business Process Management) BPM systems. One of the most fundamental characteristics required in the BPM world is the ability to support the management of dynamic business change leading through:

- Late bindings i.e. to introduce flexibility to the run time environment
- Rules engines i.e. to facilitate complex expression evaluation independently of the core process specification
- Adaptive processes i.e. to facilitate dynamic change during execution

Figure 1.7 illustrates a BPM component model, derived from a traditional view of the development and subsequent execution of a business process. The top half of the diagram illustrates the derivation of a process model along with its service delivery characteristics. The lower half illustrates the enactment of the process in this environment. The first step is to prepare a conceptual model of the program. Here one defines a model at a conceptual level and is generally prepared by business analyst. The executable model is

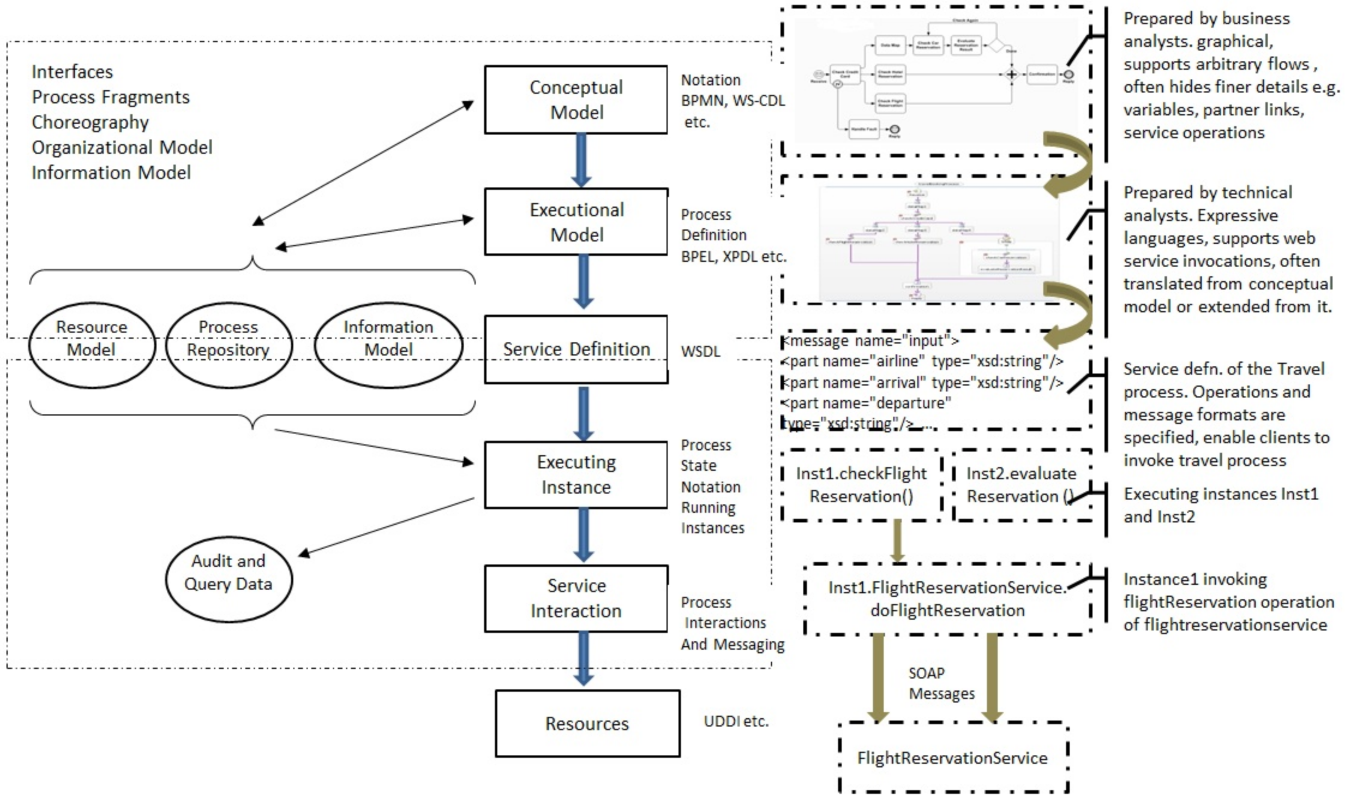

Interfaces
Process Fragments
Choreography
Organizational Model
Information Model
Conceptual Model
Notation BPMN, WS-CDL etc.
Executional Model
Process Definition BPEL, XPDL etc.
Resource Model
Process Repository
Information Model
Service Definition
WSDL
Executing Instance
Process State Notation Running Instances
Audit and Query Data
Service Interaction
Process Interactions And Messaging
Resources
UDDI etc.
Prepared by business analysts. graphical, supports arbitrary flows , often hides finer details e.g. variables, partner links, service operations
Prepared by technical analysts. Expressive languages, supports web service invocations, often translated from conceptual model or extended from it.
<message name="input">
<part name="airline" type="xsd:string"/>
<part name="arrival" type="xsd:string"/>
<part name="departure" type="xsd:string"/> ....
Service defn. of the Travel process. Operations and message formats are specified, enable clients to invoke travel process
Inst1.checkFlight Reservation()
Inst2.evaluate Reservation ()
Executing instances Inst1 and Inst2
Inst1.FlightReservationService. doFlightReservation
Instance1 invoking flightReservation operation of flightreservationservice
SOAP Messages
FlightReservationService

Figure 1.7: A BPM Component Model

prepared by technical analyst who either works on the conceptual model itself by refining it or generates this through transformation of conceptual model first. The service definitions and end points are defined in a WSDL (Refer Figure 1.2). The process is then deployed on the BPM / workflow engine (the two terms are often used interchangeably these days) and is ready for execution. On receiving a request (invoked by any client) the engine instantiates the process. The external service interactions are performed using SOAP Message exchanges.

Figure 1.8 shows an example of a conceptual BPM travel booking system. The Process begins with the receipt of a request for a travel booking (Check Credit). After a check on the credit card, reservations are made for a flight (Check Flight Reservation), a hotel (Check Hotel Reservation), and a car (Check Car Reservation). The car reservation may take more than one attempt before it is successful. After all three reservations are confirmed, a reply is sent. The flight, hotel and car reservations are all made in parallel.

**Conceptual Model:** The conceptual model is concerned with the formulation of the business process in terms of business component objects and their interactions. A conceptual model typically requires some graphical tools for process definitions or some form of a choreography language to identify the valid sequences of messages and responses across and between the participating process fragments. Processes defined here are mostly abstract processes. BPMN is the most common language for conceptual modeling. BPMN

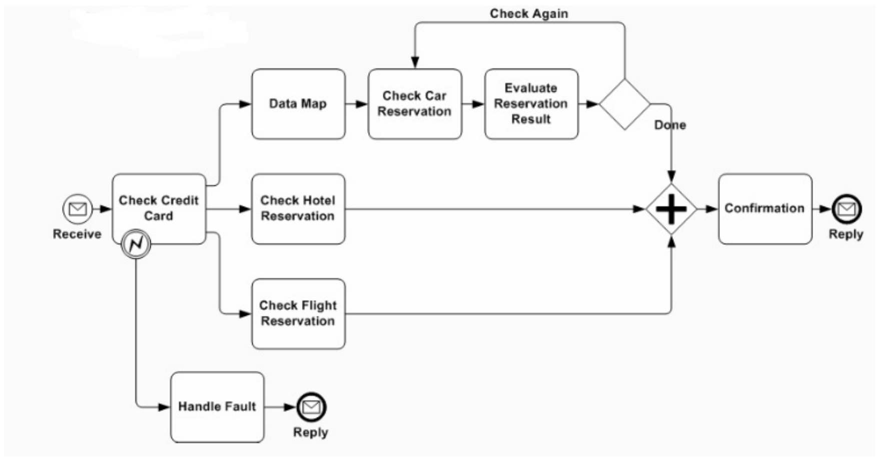


Figure 1.8: A conceptual model of travel process. Source [White, 2005]

2.0 supports choreography also. Other languages is this domain are UML Activity models and WS-CDL. It is to be noted that a conceptual model is typically prepared by a business analyst and hence the languages used are highly expressive, supports most of the patterns and allows arbitrary flows and loops etc. They allow models where resource binding etc. is not specified. A conceptual model is generally transformed into an executable language where the developers refine and add relevant information to the model. For realization, there is a need to view a high level structure to define the conversation among the various stakeholders. This can be depicted through a choreography as given in Figure 1.9. The choreography also helps in identifying the endpoint reference (EPR) (An EPR is a combination of Web services elements that define the address for the resource in a SOAP header) for the process. In Figure 1.9 the endpoint references are shown as small rectangles for the interactions between the user and the travel process.

**Executable Model:** To convert any conceptual model into an executable model requires a more detailed specification of the process in machine processable form, including not only its detailed internal structure but also its interfaces and internal resource usage. However, different BPM / workflow products have different internal representation structures of the executable model. In practice many BPM products have process design tools that are closely coupled to its execution capabilities. BPEL and XPDL are two standard languages used for executable models. The main difference between a conceptual model and an executable one is that a conceptual model is highly expressive and may support a variety of workflow patterns including arbitrary flows where the executable model depends on what the execution engine supports. Very often, people working on the conceptual model may not be aware of the runtime service component structure and hence executable model

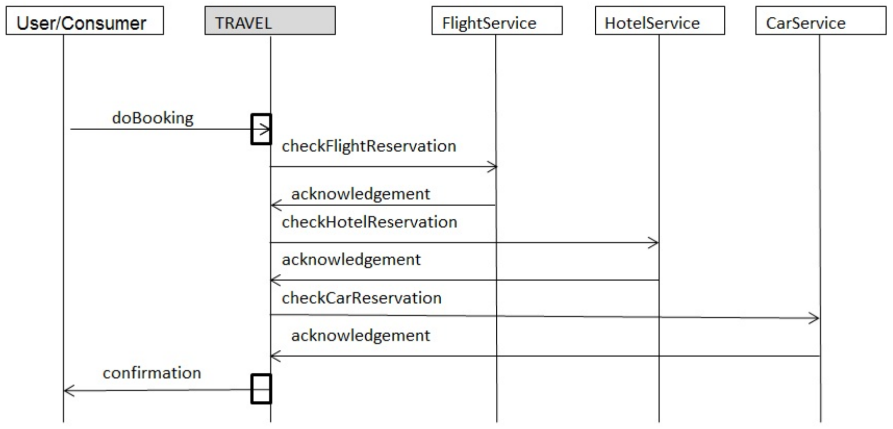


Figure 1.9: Choreography of web services in a travel workflow

may be reconfigured by the developers. The BPEL translation of the BPMN model given earlier is given in the Figure 1.10. Here the activities marked with a cross are external web-services. An extract of XML representation of this BPEL code is given in Figure 1.11.

The recent BPMN 2.0 specification has a clear executional semantics and supports both conceptual as well as the executional models. The business as well as the technical analysts can therefore work on the same model. The standardization of executional semantics enable exchange of BPMN models across BPMN 2.0 complaint platforms.

**Service Definitions:** Once the executable process is created and deployed as a service, one needs a mechanism to define it as an interface for the consumers to invoke this service. The service definition enables instantiation of the executable model(s) into operational process instances on user invocation. The service consumer invokes this via SOAP Messages which have bindings to the service as per the protocol used. The mechanism is depicted in Figure 1.12. In a web services environment, a set of supporting standards exists for defining such services, resource access points, access permissions and basic message types, etc. In other implementation environments equivalent schemes are required. WSDL is one such standard which defines the service end points.

**Service Interactions:** This represents the actual, runtime exchanges between resources associated with execution of particular process fragment. The interactions can again be internal or external. The internal interactions between the process fragments (e.g. invocation of local application resource or allocation of work to a local participant) will be regulated by the WFMS itself. External interactions however are governed by the choreographies. These are through messages e.g. SOAP, XML. Monitoring of these service interactions, especially online, is also an active area of research.

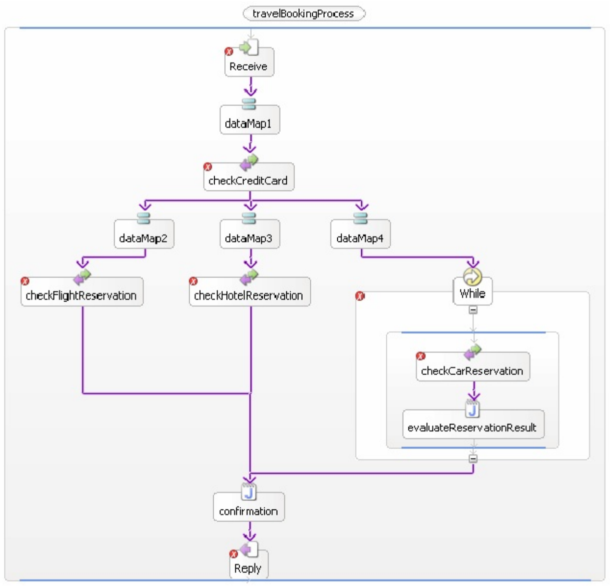


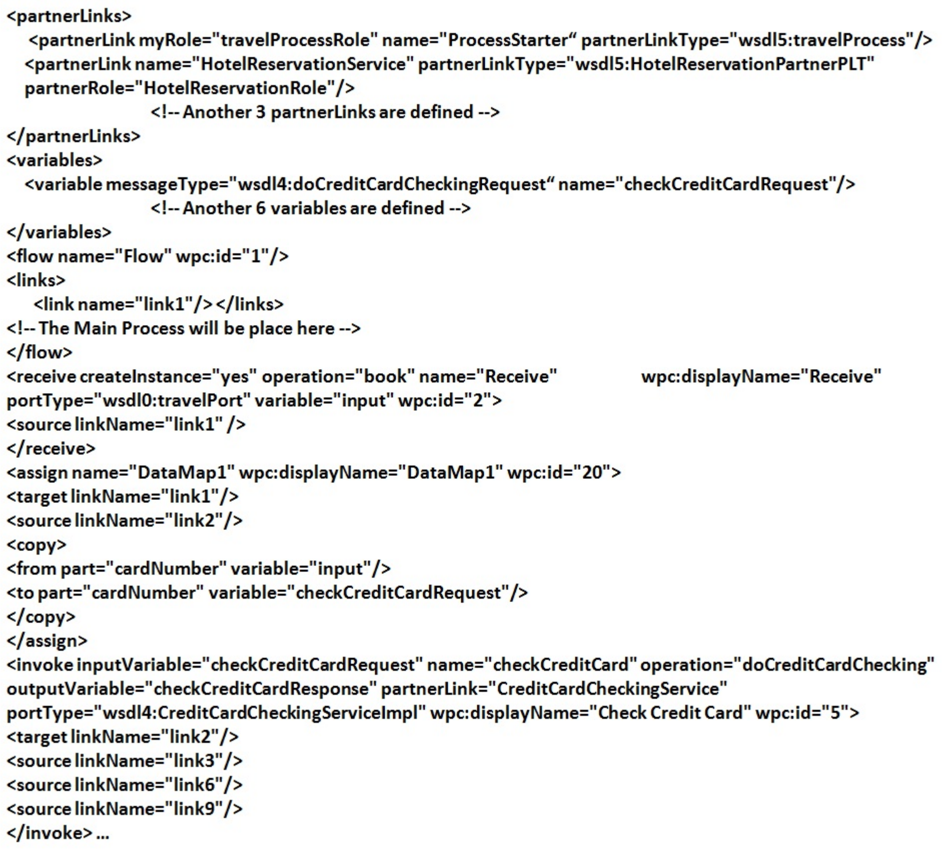
Figure 1.10: A BPEL model of the travel workflow. Source [White, 2005]

**Resource:** Resource model is one of the required components of the process definition. For internal process fragments (activities), appropriate resources are bound to these activities according to the set of rules (roles) defined earlier. These need to be flexible enough to permit late binding of resources to task. The resources could be infra-structural resources also e.g some long running computation to be allocated to a grid. Sometimes there may be a clash of resources. For external interaction one may need a mechanism to first find an appropriate resource and then assign tasks to it. With the emerging cloud computing environment resource allocation and their binding with activities in a workflow, new standards may be required in this field.

## 1.6 Challenges in the field of business process workflows

The correctness and performance of the business processes supported by the WFM are vital to the organization and hence it is important to ensure these both at design time and runtime. A workflow process definition is therefore analyzed before it is put into production for its *validation* i.e. testing whether the workflow behaves as expected; *verification*, i.e. establishing the correctness of the workflow; and *performance analysis*, i.e. evaluating the ability to meet requirements with respect to throughput times, service levels and resource utilization. Some of the key challenges in the field of business workflows are given below:

1. *Semantically enhanced service discovery:* A business process composed from many



```
<partnerLinks>
   <partnerLink myRole="travelProcessRole" name="ProcessStarter“ partnerLinkType="wsdl5:travelProcess"/>
   <partnerLink name="HotelReservationService" partnerLinkType="wsdl5:HotelReservationPartnerPLT"
   partnerRole="HotelReservationRole"/>
                  <!-- Another 3 partnerLinks are defined -->
</partnerLinks>
<variables>
   <variable messageType="wsdl4:doCreditCardCheckingRequest“ name="checkCreditCardRequest"/>
                  <!-- Another 6 variables are defined -->
</variables>
<flow name="Flow" wpc:id="1"/>
<links>
    <link name="link1"/> </links>
<!-- The Main Process will be place here -->
</flow>
<receive createInstance="yes" operation="book" name="Receive"          wpc:displayName="Receive"
portType="wsdl0:travelPort" variable="input" wpc:id="2">
<source linkName="link1" />
</receive>
<assign name="DataMap1" wpc:displayName="DataMap1" wpc:id="20">
<target linkName="link1"/>
<source linkName="link2"/>
<copy>
<from part="cardNumber" variable="input"/>
<to part="cardNumber" variable="checkCreditCardRequest"/>
</copy>
</assign>
<invoke inputVariable="checkCreditCardRequest" name="checkCreditCard" operation="doCreditCardChecking"
outputVariable="checkCreditCardResponse" partnerLink="CreditCardCheckingService"
portType="wsdl4:CreditCardCheckingServiceImpl" wpc:displayName="Check Credit Card" wpc:id="5">
<target linkName="link2"/>
<source linkName="link3"/>
<source linkName="link6"/>
<source linkName="link9"/>
</invoke> ...
```

Figure 1.11: A BPEL code of the travel workflow

services can itself be exposed as a service. The main challenge of service discovery is using “automated” means to accurately discover these services with minimal user involvement. This requires analysis of the semantics for both the service provider and requester, adding semantic annotations and including descriptions of QoS characteristics in the Web Ontology Language (or other semantic markup language) to service definitions in WSDL and then registering these descriptions. Achieving automated service discovery requires explicitly stating needs in some formal request language.

2. *Security:* Validating the security aspects in workflow applications requires a full system approach for an end-to-end security solution. This is especially true for B2B workflows. Security is a wide area which concerns the internal as well as external aspects. The internal aspects relate to access permissions, identity management, digital signatures and authentication mechanism for process access within the organization. Definition of appropriate roles and organization rules for substitutions may become very complex. The external requirements are security issues related to web services. These relate to SOAP message signing, non-repudiation, encryption and attaching security tokens to ascertain senders identity. Since workflows involve flow of data across organizations, ensuring a secure access mechanism for the interest and

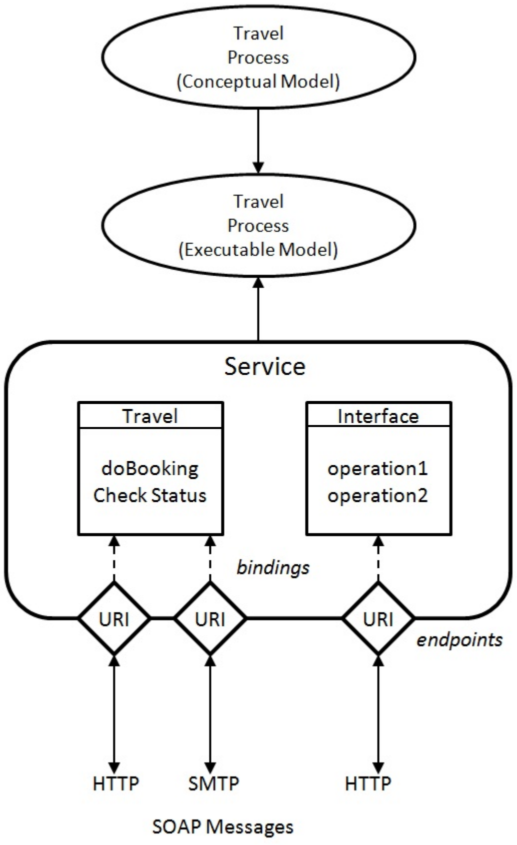


Figure 1.12: Service definition for endpoint references in BPM

requirements of all parties involved is important.

3. *Adaptive processes:* Services and processes need to equip themselves with adaptive capabilities so that they can continually change themselves to respond to environmental demands without compromising operational, QoS and financial efficiencies. The Autonomic computing techniques are required for dynamic service compositions that are self-configuring, self-optimizing, self-healing, and self-adapting.

4. *QoS-aware service compositions*: Service compositions must be QoS-aware i.e., understand the policies, performance levels, security requirements, service-level agreement (SLA) objectives, and so on of the providers. For example, if a new business process adopts a Web services security standard from WS-Security, the client not only needs to know this but also if the services in the business process actually require WS-Security, what kind of security tokens they are capable of processing, and which one they prefer etc.

5. *Process Engineering:* This refers to the methods and models adopted by any organization for their process development. Business workflows require a service-oriented

engineering methodology that enables modeling and specification of the business environment, including Business Level Objectives (BLOs) of their goals and policies; translating models into service designs; deploying the service system; and testing and managing the deployment.

6. *Service governance:* Due to the cross-organizational nature of end-to-end business processes composed from various service fragments that different organizations maintain separately, service governance is a challenging issue. Such processes can function properly and efficiently only if the services are effectively governed for compliance with QoS and the overall policy requirements. The services must therefore meet the functional and QoS objectives within the context of the enterprises within which they operate. For this it is important to be able to monitor them dynamically at runtime.

7. *Inter-operability: standards and implementation strategies*: High inter-operability is imperative for B2B applications because transactions require cooperative workflows. In an e-commerce environment, it is conceivable that workflows with various levels of automation must coexist. Some workflow systems can be completely automated with possibly sophisticated software agents, while other workflow systems may be completely manually coordinated.

8. *Transaction Management:* Since the workflow applications are distributed, typically long running and have a large number of wait states, ensuring the atomicity of a transaction running across enterprises is very important. Compensation and exception handling without manual intervention is one way to handle this and most of the languages support this in some form or the other.

## 1.7 Motivation

Business process workflows is a promising technology for today's enterprises to strengthen and streamline their core business processes and achieve profits like never before. Newer models of the business leveraging this technology are now emerging. Realizing such workflows is, however, challenging due to multiple stake holders involved as well as technical issues like inter-operability, security, process specification, process monitoring as well as the overall dynamism involved in the execution environment. Among the many stated research challenges in this field we are particularly interested in the synthesis and runtime monitoring of cross organizational workflows. In our view, these are the most important aspects holding wide scale implementations of such process workflows. The clouds and grid computing environment are now equipped to provide the necessary infrastructure for such workflows, however, unless the organizations have some means to monitor and ensure correctness of their process executions they will be hesitant to move in this direction. The motivation of this research is therefore to develop a framework that enables realization of an efficient, correct and inter-operable workflow system. The focus area is *Engineering* and *Service governance* to enable synthesis and runtime monitoring of business process workflows.

## 1.8 Objectives

The main objectives of this thesis are given below:

1. *Synthesizing a business workflow from a conceptual model in BPMN:* Given an executional language we aim to provide a framework to transform a conceptual model in BPMN to an executional model by applying model transformation techniques. With this, the business analysts will have the capability to express their solutions in terms of service interactions for both control and data flow centric workflows. It will also enable creation/simulation of mock scenarios for verification/validation/debugging in an integrated way.

2. *Runtime Monitoring of a workflow process instance/s:* Further, an executing business process relies heavily on external web services, but it does not have any direct control on the individual web-services used in its composition, which may be modified by the service providers, resulting in erroneous results. The administration and monitoring tools available in the current workflow systems are insufficient with capabilities to monitor the process state and other basic admin features only. We aim to provide a monitoring framework that enables dynamic monitoring of business process workflows on a real-time basis.

3. *Adaptive business processes:* The workflows often depend on data that is not available locally and require inputs from computations occurring at runtime. It also needs to adapt itself with the changing environment. Our objective is to integrate our monitoring framework with the workflow process to enable such adaptations.

4. *Monitoring SLAs*: Very often organizations enter into a Service Level Agreements (SLAs) with their partners in order to guarantee their process behavior. We aim to providea monitoring framework for monitoring such SLAs.

## 1.9 Organization of rest of the thesis

The rest of the thesis chapters are organized as follows:

1. *Literature review:* This chapter provides a comparison of the currently available workflow products from both open source and commercial domain, the languages they support and their features, capabilities and the limitations. This is followed by a literature survey on the various approaches for runtime monitoring of business process workflows, some research gaps and an overview of the approach taken.

2. *Background:* This chapter provides a description of the language BPMN and Orc, which forms the basis of the synthesis tool named Bpmn2Orc. It also provides the background on MSC and SL, and the trace semantics that is necessary for the runtime monitoring framework named Wf_Sla_Mon

3. *Realizing workflow patterns in Orc:* In this chapter, each of the forty three workflow control-flow patterns is analysed and its implementation is provided using the language Orc. The results are then compared with the patterns supported by BPMN and BPEL.

4. *A MDA based approach for Synthesizing Process Workflows:* Here, the system Bpmn2Orc is described which transforms a BPMN model to Orc for execution. It first provides the mapping of BPMN constructs with that of Orc and then discusses the implementation algorithm along with the illustration with a case study.

5. *Runtime Monitoring Framework:* In this chapter, the approach for runtime monitoring, its key component structure and methodology is discussed using an example BPMN system, EasyTravel and its SLA.

6. *Monitor Specification and Realization:* The chapter details the property specification using SL and MSC, the detail of the algorithms for generating monitors and their integration with Orc workflow engine.

7. Wf_Sla_Mon*: GUI Capabilities and Experimental Evaluation:* This chapter includes some implementation details, screenshots and the results of the performance evaluation tests performed.

8. *Conclusions, Specific Contributions, Limitations and Future Work:* The main contribution of the thesis, conclusions, limitations and some future scope of work is discussed in this chapter.

# Chapter 2

# Literature Review

This chapter provides a comparison of the some contemporary BPM systems from both open source and commercial domain, the languages they support, their features, capabilities and the limitations. Later, various model transformation methods and their applications are listed. This is followed by a literature survey on the related approaches for runtime monitoring of cross organizational business process workflows. After discussing the research gaps in this area, the approach taken for this thesis is discussed.

## 2.1 Current BPM Systems

A list of some of the BPM systems (from both open source and commercial domain) and their providers is given in Table 2.1. This list is just a snapshot and far from complete. The number of suppliers offering BPM solutions is estimated to be around 250. This large number itself indicates that this is a demanding area and these systems are expected to play a major role in the future. Besides the specialized "workflow" or "BPM" systems (the terms are used interchangeably these days), most Enterprise Resource Planning (ERP) systems like SAP, Baan and JD Edwards also have a workflow engine incorporated. These workflow engines cannot be used as stand alone BPM systems and requires a third party integration as well as a tight coupling with their existing ERP system. Despite the large number of products and suppliers, the number of workflows actually in production is relatively limited. Some of the reasons for their relatively low acceptance in spite of the huge demand is:

- the technology is new and many of the issues are not addressed in totality resulting in limited functionality and unsatisfactory reliability.

- successful implementations generally require some re-engineering of the organizations in terms of its functioning. This leads to the inevitable problem of "resistance to change".

- despite the efforts of WFMC, standards with respect to functionality and system linking are lacking. For example, many systems use ad hoc drawing techniques to

Table 2.1: List of some BPM Products and their Suppliers

| Name | Supplier |
|---|---|
| Action Workflow | Action Technologies Inc. |
| Activiti | Open Source |
| jBPM | Open Source |
| COSA | Ley GmbH |
| Flo Ware | BancTec-Plexus, Japan |
| InConcert | TIBCO/ InConcert |
| Income | Promati |
| Open Workflow | Wang |
| PowerFlow | Optica Imaging Solutions |
| Process Weaver | Cap Gemini Innovation |
| SAP Business Workflow | SAP AG |
| Oracle BPEL Manager | Oracle Corporation |
| WebFlow | Cap Gemini Innovation |
| WorkVision | IA Corporation |
| Ultimus | Ultimus |
| Staffware | StaffWare |
| TeamWARE | TeamWare |
| Websphere Process Manager | IBM |

specify processes. One of the drawbacks of this is that it is difficult to exchange process descriptions between different supplier systems.

- since BPM is a complex and quite wide, there is no single BPM product which meets the requirement of complex rules of resource allocations, document management, security, monitoring, control flow etc. Hence, different products target different kind/s (functional domain) of organization.

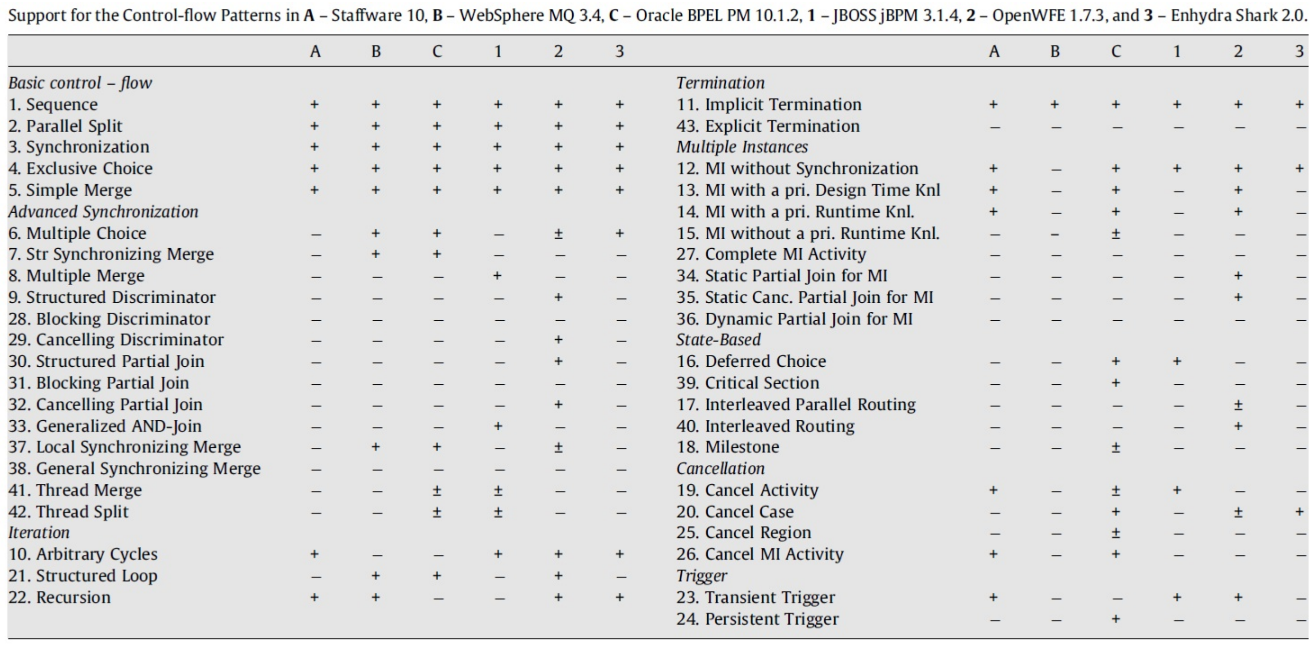
In order to get an insight of the current products in the market these days, a comparison of some of these products against the key features discussed in the previous chapter is provided below. The products that have been considered here are *Staffware*, *WebSphere MQ*, *Oracle BPEL Process Manager*, *jBPM*, *OpenWFE*, *Enhydra Shark*, *Activiti* and *YAWL*. Most of the information gathered is from the analysis given by Aalst et al. in [van der Aalst and vanHee, 2004] and [Wohed et al., 2009] or the individual product website wherever available. Some key information for commercial products *COSA* and *InConcert* is also included, though, due to lack of documentation a detailed comparison in terms of workflow patterns support is not included. Oracle BPEL and Websphere MQ being commercial products also provide limited information about the core implementation details.

**Process Definition**: *YAWL* is a BPM/Workflow system, based on a concise and powerful modeling language (Yet Another Workflow Language), that handles complex data transformations, and full integration with organizational resources and external Web Services. It provides a graphical tool for process definition in YAWL which is executable. *jBPM* is a open source initiative and has a process definition tool called jBPM Graphical Process Designer (GPD) which is a plugin to Eclipse. It provides support for defining processes in jBPM Process Definition Language (jPDL) in both: graphical as well as XML format. jPDL is a executable language and is directly executed by the jBPM workflow engine. *OpenWFE* is also a open source product which has a web-based workflow design environment called "Droflo" to define process models graphically. These are then translated to OpenWFE internal process definition language in XML format. *Enhydra Shark* has a Process Definition tool called "Together WorkFlow Editor" (TWE) which uses XPDL as a process language. *Oracle BPEL PM* as well as *IBM Websphere MQ* Series use some variant of BPEL core language and/or BPEL extensions. *Activiti* is an enriched jBPM (the core development team is the same) and supports process definitions in BPMN 2.0. *COSA Network Edition* (CONE) by COSA is a process definition tool for defining and revising processes. Here, Petri nets are used to illustrate the process.

**Resource Allocation**: *jBPM* identity component allows definition of organizational information, such as users, groups and roles to which different tasks can be assigned. Currently the definition of this information is done through standard SQL insert statements directly into the workflow database. *OpenWFE* uses a web-interface for resource management, called UMAN (for User Management). Users are either human participants or automated agents. Automated agents are implemented in a language like Java or Python and defined as users in a workflow. *Enhydra Shark* has no separate tool for resource allocation but supports resource patterns as given in Figure 2.3. *YAWL* provides a graphical resource allocation framework and has maximum support for the resource patterns. All commercial tools provide tools for resource allocation and user management.

**Administration and Monitoring Support**: Most of the providers combine resource management with monitoring. *jBPM* console web application has a web based workflow client whereby users can initiate and execute processes. It also has an administration and monitoring tool where users can observe and intervene in currently executing process instances. Enhydra Shark has a Management Console, TWS Admin, which is a workflow client as well as an administration and monitoring tool. This is however simple monitoring related to starting/stopping or viewing an executing process instance.

**Workflow Patterns**: The core workflow engine of *jBPM*, *Enhydra Shark* and *Activiti* is in Java and *OpenWFE* is in Ruby. Aalst et al [Van Der Aalst et al., 2003] have provided a set of workflow patterns, that provide a taxonomy of recurring concepts and constructs relevant to the workflow and business process management area and forms the basis of evaluating workflow systems. These patterns have been categorized as control-flow, data-flow and resource-flow patterns. Resource-flow patterns capture the various ways in which resources are represented and utilized in workflows. Data-flow patterns capture the various ways in which data is represented and utilized in workflows and control-flow patterns provides the patterns from a control flow perspective. A table comparing some of these systems as analysed by Aalst et al. in [Wohed et al., 2009] is reproduced in Figures 2.1 to 2.3. In addition, Aalst et al. have also provided an analysis of BPMN, BPEL and YAWL support for various workflow patterns in [Wohed et al., 2006], [Wohed et al., 2002], [Wohed et al., 2003] and [Hofstede, 2005] respectively. YAWL has maximum support for workflow as well as resource patterns and is also implemented in some "Health care", "Product recall" scenarios.


Support for the Control-flow Patterns in **A** – Staffware 10, **B** – WebSphere MQ 3.4, **C** – Oracle BPEL PM 10.1.2, **1** – JBOSS jBPM 3.1.4, **2** – OpenWFE 1.7.3, and **3** – Enhydra Shark 2.0.

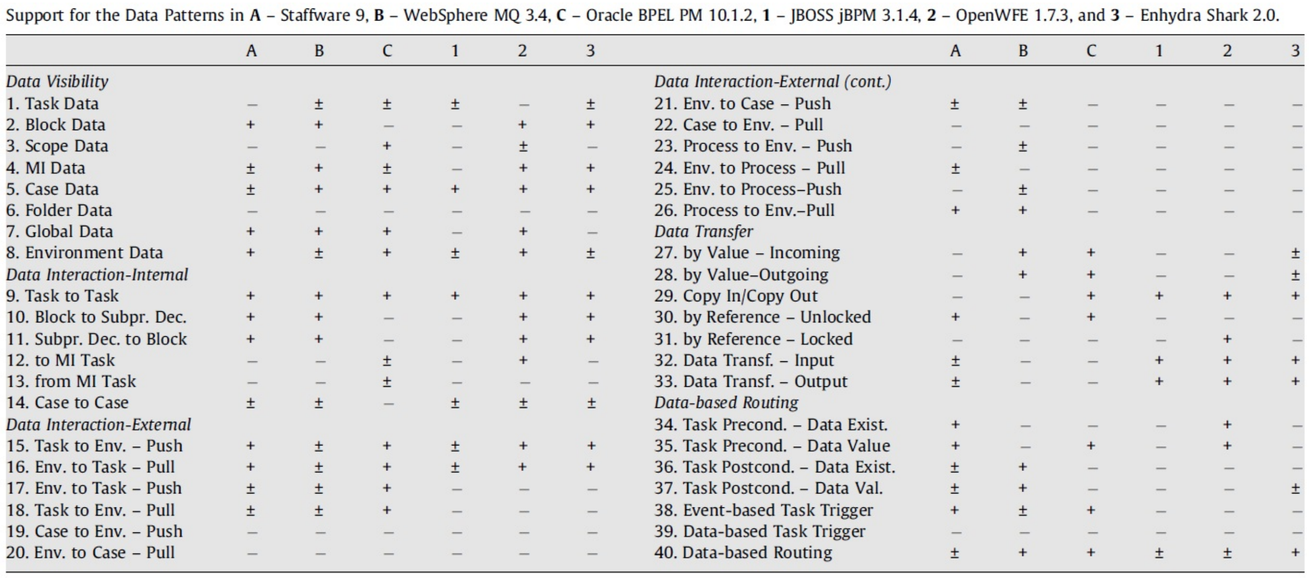
| | A | B | C | 1 | 2 | 3 | | A | B | C | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Basic control – flow* | | | | | | | *Termination* | | | | | | |
| 1. Sequence | + | + | + | + | + | + | 11. Implicit Termination | + | + | + | + | + | + |
| 2. Parallel Split | + | + | + | + | + | + | 43. Explicit Termination | – | – | – | – | – | – |
| 3. Synchronization | + | + | + | + | + | + | *Multiple Instances* | | | | | | |
| 4. Exclusive Choice | + | + | + | + | + | + | 12. MI without Synchronization | + | – | + | + | + | + |
| 5. Simple Merge | + | + | + | + | + | + | 13. MI with a pri. Design Time Knl | + | – | + | – | + | – |
| *Advanced Synchronization* | | | | | | | 14. MI with a pri. Runtime Knl. | + | – | + | – | + | – |
| 6. Multiple Choice | – | + | + | – | ± | + | 15. MI without a pri. Runtime Knl. | – | – | ± | – | – | – |
| 7. Str Synchronizing Merge | – | + | + | – | – | – | 27. Complete MI Activity | – | – | – | – | – | – |
| 8. Multiple Merge | – | – | – | + | – | – | 34. Static Partial Join for MI | – | – | – | – | + | – |
| 9. Structured Discriminator | – | – | – | – | + | – | 35. Static Canc. Partial Join for MI | – | – | – | – | + | – |
| 28. Blocking Discriminator | – | – | – | – | – | – | 36. Dynamic Partial Join for MI | – | – | – | – | – | – |
| 29. Cancelling Discriminator | – | – | – | – | + | – | *State-Based* | | | | | | |
| 30. Structured Partial Join | – | – | – | – | + | – | 16. Deferred Choice | – | – | + | + | – | – |
| 31. Blocking Partial Join | – | – | – | – | – | – | 39. Critical Section | – | – | + | – | – | – |
| 32. Cancelling Partial Join | – | – | – | – | + | – | 17. Interleaved Parallel Routing | – | – | – | – | ± | – |
| 33. Generalized AND-Join | – | – | – | + | – | – | 40. Interleaved Routing | – | – | – | – | + | – |
| 37. Local Synchronizing Merge | – | + | + | – | ± | – | 18. Milestone | – | – | ± | – | – | – |
| 38. General Synchronizing Merge | – | – | – | – | – | – | *Cancellation* | | | | | | |
| 41. Thread Merge | – | – | ± | ± | – | – | 19. Cancel Activity | + | – | ± | + | – | – |
| 42. Thread Split | – | – | ± | ± | – | – | 20. Cancel Case | – | – | + | – | ± | + |
| *Iteration* | | | | | | | 25. Cancel Region | – | – | ± | – | – | – |
| 10. Arbitrary Cycles | + | – | – | + | + | + | 26. Cancel MI Activity | + | – | + | – | – | – |
| 21. Structured Loop | – | + | + | – | + | – | *Trigger* | | | | | | |
| 22. Recursion | + | + | – | – | + | + | 23. Transient Trigger | + | – | – | + | + | – |
| | | | | | | | 24. Persistent Trigger | – | – | + | – | – | – |

Figure 2.1: Workflow Control-flow Patterns support in various BPM products. Source [Wohed et al., 2009]

As can be seen, various products support workflows with some or the other variant of the standards based languages like BPEL, BPMN or have their own proprietary system. Standards are emerging and newer products as well as the newer versions of these existing products are continuously evolving. While BPEL is commonly used for execution modeling, BPMN is the language of choice for process specification. YAWL is based on Petri-nets and provides a common modeling framework.

## 2.2 Model Transformation Techniques

The Model-Driven Architecture (MDA) by Object Management Group (OMG) aims to automate the generation of platform-specific models from platform independent models. The basic MDA pattern involves defining a platform independent model (PIM) and its



Support for the Data Patterns in **A** – Staffware 9, **B** – WebSphere MQ 3.4, **C** – Oracle BPEL PM 10.1.2, **1** – JBOSS jBPM 3.1.4, **2** – OpenWFE 1.7.3, and **3** – Enhydra Shark 2.0.

| | A | B | C | 1 | 2 | 3 |
|---|---|---|---|---|---|---|
| *Data Visibility* | | | | | | |
| 1. Task Data | – | ± | ± | ± | – | ± |
| 2. Block Data | + | + | – | – | + | + |
| 3. Scope Data | – | – | + | – | ± | – |
| 4. MI Data | ± | + | ± | – | + | + |
| 5. Case Data | ± | + | + | + | + | + |
| 6. Folder Data | – | – | – | – | – | – |
| 7. Global Data | + | + | + | – | + | – |
| 8. Environment Data | + | ± | + | ± | + | ± |
| *Data Interaction-Internal* | | | | | | |
| 9. Task to Task | + | + | + | + | + | + |
| 10. Block to Subpr. Dec. | + | + | – | – | + | + |
| 11. Subpr. Dec. to Block | + | + | – | – | + | + |
| 12. to MI Task | – | – | ± | – | + | – |
| 13. from MI Task | – | – | ± | – | – | – |
| 14. Case to Case | ± | ± | – | ± | ± | ± |
| *Data Interaction-External* | | | | | | |
| 15. Task to Env. – Push | + | ± | + | ± | + | + |
| 16. Env. to Task – Pull | + | ± | + | ± | + | + |
| 17. Env. to Task – Push | ± | ± | + | – | – | – |
| 18. Task to Env. – Pull | ± | ± | + | – | – | – |
| 19. Case to Env. – Push | – | – | – | – | – | – |
| 20. Env. to Case – Pull | – | – | – | – | – | – |
| *Data Interaction-External (cont.)* | | | | | | |
| 21. Env. to Case – Push | ± | ± | – | – | – | – |
| 22. Case to Env. – Pull | – | – | – | – | – | – |
| 23. Process to Env. – Push | – | ± | – | – | – | – |
| 24. Env. to Process – Pull | ± | – | – | – | – | – |
| 25. Env. to Process–Push | – | ± | – | – | – | – |
| 26. Process to Env.–Pull | + | + | – | – | – | – |
| *Data Transfer* | | | | | | |
| 27. by Value – Incoming | – | + | + | – | – | ± |
| 28. by Value–Outgoing | – | + | + | – | – | ± |
| 29. Copy In/Copy Out | – | – | + | + | + | + |
| 30. by Reference – Unlocked | + | – | + | – | – | – |
| 31. by Reference – Locked | – | – | – | – | + | – |
| 32. Data Transf. – Input | ± | – | – | + | + | + |
| 33. Data Transf. – Output | ± | – | – | + | + | + |
| *Data-based Routing* | | | | | | |
| 34. Task Precond. – Data Exist. | + | – | – | – | + | – |
| 35. Task Precond. – Data Value | + | – | + | – | + | – |
| 36. Task Postcond. – Data Exist. | ± | + | – | – | – | – |
| 37. Task Postcond. – Data Val. | ± | + | – | – | – | ± |
| 38. Event-based Task Trigger | + | ± | + | – | – | – |
| 39. Data-based Task Trigger | – | – | – | – | – | – |
| 40. Data-based Routing | ± | + | + | ± | ± | + |

Figure 2.2: Workflow Data-flow Patterns support in various BPM products. Source [Wohed et al., 2009]

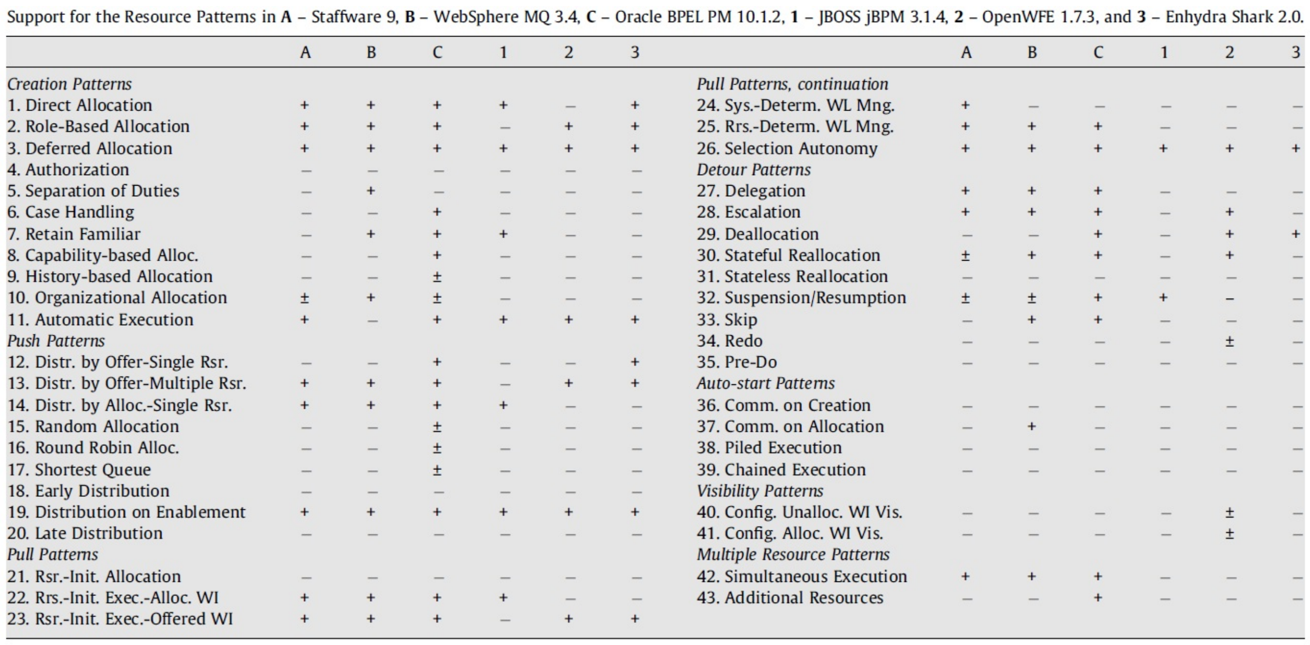
automated mapping to one or more platform-specific models (PSMs). In the context of BPM, model transformations are required to convert a conceptual model into executable model. We discuss and categorize some of the model transformation approaches here. These are broadly classified into two types: "Model to Text" and "Model to Model"

### 2.2.1 Model To Text

Model to Text (M2T) includes non code artifacts like XML as well as the code (referred as Model To Code). There are two kinds of model to text approaches:

1. *Visitor based approach*: A very basic code generation approach consists in providing some visitor mechanisms to traverse the internal representation of a model and generate code as a text stream. An example of this approach is *Jamda*, which is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism to generate code.

2. *Template based approach*: In a template based approach, "templates" are used to generate the code. A template usually consists of the target text containing splices of meta code to access information from the source and to perform code selection and iterative expansion. It closely resembles the target model and hence it results in more accurate code generation. Tools like Jet, OptimalJ, XDE, and the Model



Support for the Resource Patterns in **A** – Staffware 9, **B** – WebSphere MQ 3.4, **C** – Oracle BPEL PM 10.1.2, **1** – JBOSS jBPM 3.1.4, **2** – OpenWFE 1.7.3, and **3** – Enhydra Shark 2.0.

| | A | B | C | 1 | 2 | 3 | | A | B | C | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Creation Patterns* | | | | | | | *Pull Patterns, continuation* | | | | | | |
| 1. Direct Allocation | + | + | + | + | – | + | 24. Sys.-Determ. WL Mng. | + | – | – | – | – | – |
| 2. Role-Based Allocation | + | + | + | – | + | + | 25. Rrs.-Determ. WL Mng. | + | + | + | – | – | – |
| 3. Deferred Allocation | + | + | + | + | + | + | 26. Selection Autonomy | + | + | + | + | + | + |
| 4. Authorization | – | – | – | – | – | – | *Detour Patterns* | | | | | | |
| 5. Separation of Duties | – | + | – | – | – | – | 27. Delegation | + | + | + | – | – | – |
| 6. Case Handling | – | – | + | – | – | – | 28. Escalation | + | + | + | – | + | – |
| 7. Retain Familiar | – | + | + | + | – | – | 29. Deallocation | – | – | + | – | + | + |
| 8. Capability-based Alloc. | – | – | + | – | – | – | 30. Stateful Reallocation | ± | + | + | – | + | – |
| 9. History-based Allocation | – | – | ± | – | – | – | 31. Stateless Reallocation | – | – | – | – | – | – |
| 10. Organizational Allocation | ± | + | ± | – | – | – | 32. Suspension/Resumption | ± | ± | + | + | – | – |
| 11. Automatic Execution | + | – | + | + | + | + | 33. Skip | – | + | + | – | – | – |
| *Push Patterns* | | | | | | | 34. Redo | – | – | – | – | ± | – |
| 12. Distr. by Offer-Single Rsr. | – | – | + | – | – | + | 35. Pre-Do | – | – | – | – | – | – |
| 13. Distr. by Offer-Multiple Rsr. | + | + | + | – | + | + | *Auto-start Patterns* | | | | | | |
| 14. Distr. by Alloc.-Single Rsr. | + | + | + | + | – | – | 36. Comm. on Creation | – | – | – | – | – | – |
| 15. Random Allocation | – | – | ± | – | – | – | 37. Comm. on Allocation | – | + | – | – | – | – |
| 16. Round Robin Alloc. | – | – | ± | – | – | – | 38. Piled Execution | – | – | – | – | – | – |
| 17. Shortest Queue | – | – | ± | – | – | – | 39. Chained Execution | – | – | – | – | – | – |
| 18. Early Distribution | – | – | – | – | – | – | *Visibility Patterns* | | | | | | |
| 19. Distribution on Enablement | + | + | + | + | + | + | 40. Config. Unalloc. WI Vis. | – | – | – | – | ± | – |
| 20. Late Distribution | – | – | – | – | – | – | 41. Config. Alloc. WI Vis. | – | – | – | – | ± | – |
| *Pull Patterns* | | | | | | | *Multiple Resource Patterns* | | | | | | |
| 21. Rsr.-Init. Allocation | – | – | – | – | – | – | 42. Simultaneous Execution | + | + | + | – | – | – |
| 22. Rrs.-Init. Exec.-Alloc. WI | + | + | + | + | – | – | 43. Additional Resources | – | – | + | – | – | – |
| 23. Rsr.-Init. Exec.-Offered WI | + | + | + | – | + | + | | | | | | | |

Figure 2.3: Workflow Resource-flow patterns support in various BPM products. Source [Wohed et al., 2009]

to Text (M2T) transformation project under the Eclipse Modeling Project, help in transforming models into text. However this approach is useful when both source and target models have the same meta-model.

## 2.2.2 Model To Model

Model-to-Model (M2M) transformations translate between source and target models, which can be instances of the same or different meta-models. M2M transformations are required to bridge the large abstraction gaps between PIMs and PSMs; it is easier to generate intermediate models rather than go straight to the target PSM. The different types of Model to Model approaches are:

1. *Direct Manipulation approach*: In this approach, the source model is itself modified directly. Users have to implement transformation rules and their scheduling from scratch using a programming language such as Java, an internal model representation plus some API to manipulate it. They are usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations (e.g., abstract class for transformations). Examples

of this approach include Jamda and implementing transformations directly against some MOF-compliant API (e.g., JMI).

2. *Relational approach*: In a relational approach one needs to state the source and target element types of a relation and specify it using constraints. In its pure form, such specification is non-executable. However, declarative constraints can be given executable semantics, such as in logic programming which, with its unification-based matching, search, and backtracking seems a natural choice to implement the relational approach. Here predicates can be used to describe the relations. Relational approach is thus declarative and mainly focuses on mathematical relations between source and target models. Predicates and constraints are used to define mathematical relationships.

3. *Graph Transformation based approach*: This category is based on the theoretical work on graph transformations. They operate on typed, attributed and labeled graphs e.g. representing UML like models. A graph transformation approach consists of LHS and RHS rules in the form of graph patterns. The LHS pattern consists of a subgraph and condition which is matched in the model being transformed and replaced by the RHS pattern in place.

4. *Structure Driven approach:* This is a two phased approach. The first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references in the target. The framework determines the scheduling and application strategy and users as such are only concerned with providing the transformation rules.

In addition to these approaches, there are hybrid approaches which use a combination of these approaches. In practise hybrid approach is most appropriate as a single approach is useful only for small transformations.

## 2.3 Related work in monitoring and verification of workflows

There have been several approaches and works for developing runtime monitors for business process workflows each focusing at different aspects like nature of verifiable properties, timeliness, granularity, etc. In some approaches the process is temporarily stopped while the assertion is verified while in others the verification is done via a parallel thread. Recovery techniques also range from simplistic logging or sending alerts to more complicated ones where the process execution can be modified at runtime. Some of these approaches are summarized here.

Aalst et al.[Aalst et al., 2008] have addressed the problem of conformance checking of services wherein the service message logs created by SOAP Messages are checked for conformance against the BPEL abstract process specification by converting the BPEL process into a Petri-net and subsequently WF-Net. Conformance checking is done against two parameters: *fitness* and *appropriateness* where "fitness" is confirmed if the Petri net can generate each trace in the log and "appropriateness" is confirmed if the Petri net is the simplest one explaining the observed behavior. Thus, over-fitting and under-fitting models are avoided. They have shown that specifications in terms of abstract BPEL can be mapped onto Petri nets and the SOAP messages exchanged between the various services can be mapped onto the MXML log format. Given a set of messages and an abstract BPEL specification, conformance checking of these parameters have been demonstrated. If the observed behavior does not match the specified behavior, the deviations can be shown in both the log and the model. They have used a case study utilizing Oracle BPEL as a process engine, and demonstrated that it is indeed applicable using current technology.

Spanoudakis et al.[Mahbub and Spanoudakis, 2004] have given a method for validating behavior properties expressed using the first order temporal logic language of event calculus. These properties are automatically extracted from the specification of its composition process in BPEL4WS or assumed that system providers can specify in terms of events extracted from this specification. Properties are checked using a variant of techniques for checking integrity constraints against temporal deductive databases. It supports monitoring of three different types of deviation from requirements, namely: inconsistencies with respect to recorded and expected system behavior, and unjustified system behavior.

Moser et al. [Moser et al., 2008] have presented a non intrusive approach based on AOP techniques for BPEL processes. They focus on Quality of Service (QoS) attributes and dynamic (runtime) replacement of partner web services not meeting the QoS requirements. For replacements the processes need to be syntactically and semantically equivalent to the BPEL process. They capture the events by intercepting SOAP Messages. They have named their system as "VieDAME".

Baresi et al. have given an integrated monitoring framework for BPEL by unifying their DYNAMO and ASTRO approaches [Baresi et al., 2009][Baresi et al., 2010]. Where the "Dynamo" approach is more suited for synchronous check of properties on single process instances, the "Astro" is devoted to the specification of properties that span sets or classes of process instances. The unification is at both specification and architectural level and makes it possible to have monitors which can support both time and temporal dimension and support property specification for a process instance, class instance and cross-process instance. Dynamo uses Aspect Oriented Programming (AOP) techniques to gather runtime data from a running BPEL process, and WSCoL (Web Service Constraint Language) to define the functional and non-functional properties that need to be checked during execution. Astro on the other hand, uses automatically generated and independent software modules to check properties that are defined in RTML (Run-Time Monitor specification Language). Although basic events are collected using AOP techniques, the actual analysis is performed by independent software modules that run in parallel to the process execution. Monitors are automatically generated and deployed starting from a declarative definition. The code generation produces a state-transition system that evolves on the basis of events collected at run time.

Organizations normally enter into Service Level Agreements (SLAs) with their business partners to ensure the quality and safety aspects of their processes which depend on the services provided by the providers. A significant aspect of any monitoring framework is ensuring conformance to these SLAs. SLA properties are required to be specified formally and many approaches have been proposed for the specification and runtime monitoring of SLAs. Raimondi et al.[Raimondi et al., 2008] have considered timeliness constraints, such as latency, throughput, availability and reliability which are formally specified in service level agreements and translated into timed automata. They attach time stamps to SOAP Messages and consider these messages as timed letters. SLA Violations are detected by means of executing the timed automata accepting the timed words. Similar work focusing on SLA is also found in [Sahai et al., 2002], [Morgan et al., 2005], [Keller and Ludwig, 2003], [Khaxar et al., 2009] and [Comuzzi et al., 2009].

Ludwig et al. offer an approach based on WS-Agreement defining the Cremona framework for the creation and monitoring of agreements. WS-Agreement and the framework facilitates providing QoS guarantees by (a) enabling the client to state its service capacity and QoS needs at the time of publishing (b) enabling the provider to derive resource requirements for the requested QoS level and capacity, and to prioritize allocation of resources when enough resources are not available (c) the provider to accept or reject a request by a client based on the resource situation at the time the client requests the service and (d) a runtime interface to monitor these agreements and to take agreement-level actions, e.g. changing to another provider or extending agreements if more capacity is needed.

Yuan et al. use UML Sequence Diagram 2.0 for property specification from which they derive a Finite State Automata. UML 2.0 Sequence Diagrams are sufficiently expressive for capturing safety and liveness properties [Gan et al., 2007]. By transforming these diagrams to automata, they enable conformance checking of finite execution traces against the specification. Halle et al. have used XML streaming for runtime monitoring [Hallé and Villemaire, 2009]. Properties are specified using LTL which is mapped to XQuery. They have not considered non functional properties and aggregation of metrics which are required in a typical SLA.

## 2.4 Research Gaps

Though there are many languages for BPM available in the market today as detailed in Section 2.1, each language has its own merits or demerits in terms of the features supported and the target systems it addresses. Most of the BPM languages are more control-centric rather than being data-centric. They are generally not suitable for scientific workflows and other distributed programming languages. There are limitations in terms of the workflow patterns support also. With the emergence of cloud and grid computing environments the scope of business workflows are likely to increase and they need to be equipped with capabilities that can leverage such technologies for cloud services as well.

In addition, most of the current BPM frameworks use a common language for both executional as well as conceptual models rather than having a separate language for conceptual modeling and then transforming it to a executional model. Using a common language makes the conceptual model more complicated for the business analysts. At the same time different source and target languages limits the conceptual model to the features of the target language. The workflows today need to support complex compositions and yet have an ability to be specified by business analyst community easily. A framework is thus required that allows business community to specify their processes using a language suited for conceptual modeling which can then be translated to an executional language that supports a large number of features and can be directly executed or augmented with additional details by technical analysts later. Based on the study of various BPM systems we have chosen BPMN as a conceptual modeling language as it is graphical and yet supports a large number of workflow patterns. Orc is the language of our choice for execution language.

The workflow administration and monitoring tools available in the current BPM systems have very limited capabilities. It is highly insufficient when it comes to ensuring the conformance requirements of the business process with respect to safety or liveness properties. Very often organizations enter into a Service Level Agreements (SLAs) with their partners in order to guarantee their process behavior. However, there is no mechanism to ensure that these agreements are indeed adhered to. A lot of work has been done in this area as detailed in Section 2.3. Most of the existing approaches however rely on events to be monitored being generated through code instrumentation. These statements may be inserted in the source or compiled code of a system and are therefore intrusive. Others use the message logs generated at runtime which are then evaluated for their conformance. These are based on runtime logs, they do not work on real time basis and, hence, are not ideal to support process adaptations. The current work on SLA monitoring considers SLAs to have purely quality related properties. However, SLAs these days can be quite complex, with functional as well as QoS properties. No single approach can therefore be considered as exhaustive enough, that allows dynamic monitoring as well as process adaptations. Moreover, to the best of our knowledge there is no existing work which demonstrates how a Service Level Agreement can be broken into measurable properties and dynamically monitored to allow the workflow adapt itself using a feedback mechanism. With these gaps in mind we aim to provide a system that allows synthesizing, runtime monitoring as well as adaptation of a business process workflow.

## 2.5 Approach

The approach taken towards meeting the objectives stated in Chapter 1 in view of the research gaps is as follows:

### 2.5.1 Realization of an executional model in Orc

The language Orc [Misra, 2006] and [Kitchin et al., 2009] is based on a few simple constructs and provides a clean separation between computation and orchestration and has been used to model different kinds of orchestrations and workflow patterns [Cook et al., 2006].

We first perform an analysis of the forty three workflow control flow patterns support in Orc and compare them with BPMN and BPEL. The study shows that Orc supports most of the patterns like Multi merge, Discriminator, Arbitrary cycles, and some of the Multiple instances patterns which are important for many service based systems but not directly supported by languages like BPEL. It also has the capability of realizing map-reduce enabled workflow compositions [Fei et al., 2009] and also simulations of physical phenomena of real world. For these reasons Orc is chosen as a realistic choice for a workflow executional language.

With BPMN as the language for conceptual modeling, certain transformation techniques are applied to transform it into *Orc* (the chosen executional language for workflows). The system, BPMN2ORC, uses a MDA based approach and hence extends the capabilities of the language Orc to the business analysts community and the cloud computing environments. It provides the business analysts the means to express their solutions in terms of services and their interactions for control/data flow centric workflows that may require capabilities like map-reduce also. This will enable creation/simulation of mock scenarios for such workflows and the use of verification/validation/debugging in an integrated way.

For transformation, an in depth study of language BPMN and Orc is performed resulting in a set of mappings between BPMN constructs and their equivalent Orc structures. The transformation approach is graph based where-in the XML based BPMN diagram is converted to an Orc Graph. The graph is first used to validate the BPMN diagram and later traversed as per the algorithm to generate the Orc code.

### 2.5.2 Runtime Monitoring of a workflow process instance/s

A runtime monitoring system WF_SLA_MON is developed that supports specification of both functional and non functional properties for any workflow and initiates self recovery actions in case of violation. An "Observer based approach for monitoring" where the system executes concurrently with the runtime system of the workflow engine is used. MSC and SL is used for property specification from where the observers are generated automatically as an automata. These properties can be combined to form the Service Level Agreements also. The observers are integrated with the underlying web-service engine to allow process adaptations.

The complete system has support for a web based interface starting from code synthesis, property specification, SLA composition and online monitoring. It makes use of formal techniques for developing efficient, correct and inter-operable workflows that may be control flow or data flow centric. The illustration of implementation uses a travel process and its SLA properties to be monitored.

# Chapter 3

# Background

This chapter provides the background information required for the thesis work. The chapter is divided into two parts. The first part (Section 3.1) includes a background of the language BPMN and Orc, which forms the basis of our synthesis tool Bpmn2Orc. The second part (Section 3.2) includes a background on MSC, SL and the trace semantics that is required for the runtime monitoring framework named Wf_Sla_Mon.

## 3.1 BPMN and Orc

### 3.1.1 Business Process Modeling Notation (BPMN)

BPMN provides a graphical notation for business process modeling, with an emphasis on control flow. Its objective is to support business process management for both technical and business users by providing a notation that is intuitive to business users yet able to represent complex process semantics. It defines a process using Business Process Diagram (BPD), which is in the form of a flow chart made up of BPMN elements. The BPMN elements (Refer Figure 3.1) could be

- Flow Objects
- Connecting Objects
- Swimlanes
- Artifacts

A flow object can be an *event*, an *activity* or a *gateway*. An *event* is something that "happens" during the course of a Process or a Choreography. These events affect the flow of the model and usually have a cause (trigger) or an impact (result). An event can be a *start event* (start of the process), *end event* (end of the process), a message that arrives or a specific date-time being reached during a process (intermediate/timer event). An *activity* is a generic term that stands for work to be performed within a

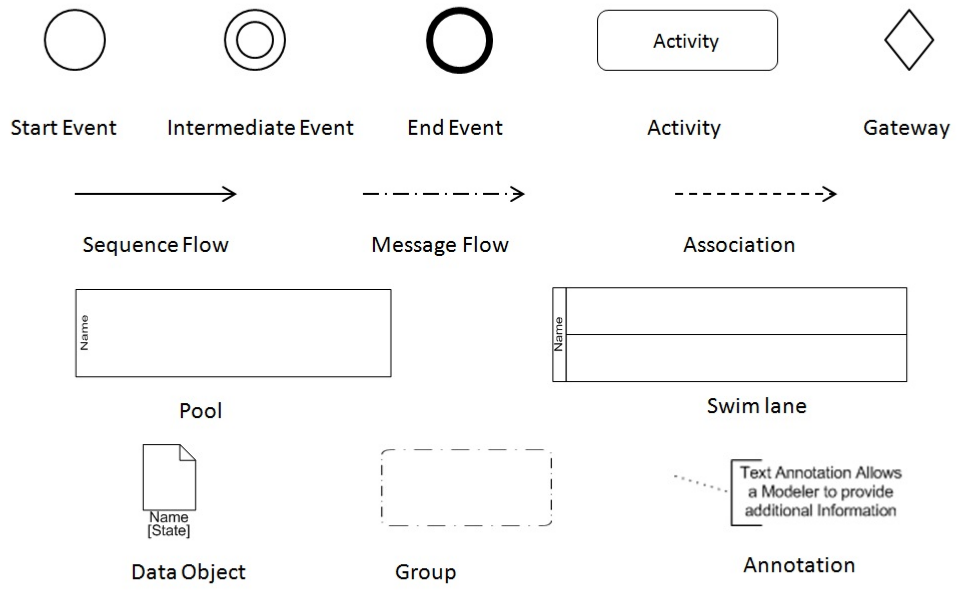


Figure 3.1: BPMN Elements

process. An Activity can be atomic or compound (sub-process). Activities and Gateways form the most important constructs for building models as they form the basis of our transformation model. Table 3.1 provides an informal semantics of BPMN gateways along with the Workflow Control flow Pattern (WCP) they represent. The types of activities that are part of any process model are: tasks, sub-processes and transactions. *Gateways* are used to control the divergence and convergence of sequence flows in a process. They determine the branching, forking, merging and joining of paths. They are represented with a diamond shape. Internal markers indicate the type of behavior control. The types of gateways include (a) exclusive split and join (b) event based split (c) inclusive split and join (d) complex split decision and join (e) parallel split and join. (Refer Figure 3.2)

Connecting objects include *sequence flows*, *message flows* and *associations*. A sequence flow links two objects in a BPD and shows the control flow relation between them. A message flow shows the flow of messages between two entities that send and receive them and are represented by a dashed line. An association is represented by a dotted line and is used to associate information with flow objects. Uncontrolled flows are those that not affected by any condition or does not pass through a Gateway.

Swimlanes include *pools* and *lanes*. A Pool is a "swimlane" and a graphical container for partitioning a set of activities from other pools, usually in the context of B2B situations. A Lane is a sub-partition within a pool and will extend the entire length of the pool, either vertically or horizontally. Lanes are used to organize and categorize activities.

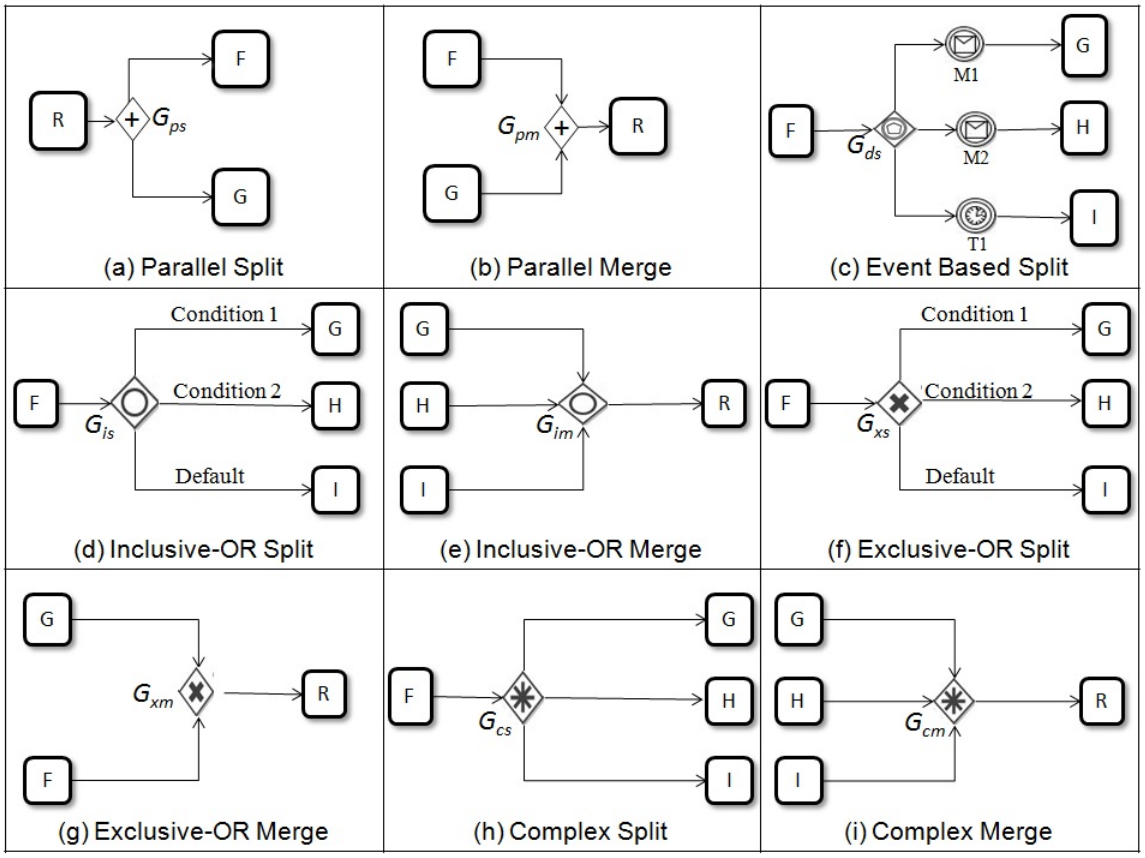


Figure 3.2: BPMN Gateways

Artifacts allow modelers and modeling tools some flexibility in extending the basic notation and in providing the ability to add additional context appropriate to a specific modeling situation. These can be *data objects*, *groups* and *annotations*. Data Objects are a mechanism to show how data is required or produced by activities. They are connected to activities through Associations. A Group is represented by a rounded corner rectangle drawn with a dashed line. The grouping can be used for documentation or analysis purposes, but does not affect the sequence-flow. Annotations are a mechanism for a modeler to provide additional information for the reader of a BPMN Diagram.

BPMN supports looping constructs. The attributes of any task or sub-processes determine if they are repeated or performed once. A small looping indicator is displayed at the bottom center of the activity. Loops can also be created by connecting a sequence flow to an upstream object.

Table 3.1: BPMN Gateway Semantics

| ELEMENT | DESCRIPTION AND WCP PATTERN SUPPORT |
|---|---|
| Parallel Split (Fork) | It is a place in the process where activities can be performed concurrently. Figure 3.2(a) shows a parallel gateway where an activity R forks two activities F and G. Represents WCP-2. |
| Parallel Merge (Join) | This gateway is used to synchronize multiple concurrent branches. Figure 3.2(b) represents this gateway where F and G are merged using this gateway and thereafter execute R. Represents WCP-3. |
| Exclusive (XOR) Split | Each activation of this gateway leads to the activation of exactly one out of the set of outgoing branches. A default node is specified so that atleast one outgoing edge is activated.(Refer Figure 3.2(c)) Represents WCP-4. |
| Exclusive (XOR) Merge | This gateway is a join where two or more paths are combined into a single path. Figure 3.2(d) represents this where F and G are joined using the XOR merge. It represents the WCP-5 and 8 pattern. |
| Inclusive Split | It represents a branching point where alternatives are based on conditional expressions contained within the outgoing sequence flows. The conditions attached to the outgoing edges can be evaluated in any order and all branches where the conditions evaluated to true are activated. The gateway is depicted in Figure 3.2(e) and represents WCP-6. |
| Inclusive Join | This gateway synchronizes a certain subset of branches out of the set of concurrent incoming branches and represents WCP-7 pattern. It is depicted in Figure 3.2(f) where F and G are merged using this join. |
| Event based split | It is a branching point where alternatives are based on an event that occurs at that point in the process. It represents WCP-16 pattern. The gateway is shown in Figure 3.2 (c) |
| Complex Choice | It is a $n$ out of $m$ gateway. All conditions of the outgoing edges are evaluated and only those that evaluate to *true* are activated. The gateway is shown in Figure 3.2 (h) |
| Complex Merge | It expresses the Structured Discriminator or a Structured Partial Join. It is to be noted that a structured partial join is a $n$ out of $m$ join whereas Structured Discriminator is *1* out of $m$ join. WCP-9 / 28. The gateway is shown in Figure 3.2 (i) |

### 3.1.2 Orc

Orc is a language of our choice for the service orchestration because of its simple constructs and the programming model that suits the requirements of a web service orchestration language. It has a mathematical foundation and supports the initial twenty workflow patterns as analysed by Misra et al. [Cook et al., 2006]. We provide here an informal introduction to Orc. For a detailed explanation the readers are encouraged to see [Misra, 2006], [Kitchin et al., 2009] and [The University of Texas, 2011].

Orc separates the computational and non-computational issues like concurrency involved in an orchestration. It assumes the presence of a set of services called *sites* which could be external web services or local functions. Sites thus provide capabilities for web services invocations. Sites could be of varying complexity, performing tasks ranging from simple arithmetic to complex data management. We use small examples to introduce the three basic combinators of Orc.

**Sequential Composition(>>)**: Consider an expression *f >x> g*. Here each value returned by *f* causes a fresh evaluation of *g*. This value can be passed to *g* via channel *x*. For e.g.

```
getRequests(date) >req> bookHotel(date,req).
```

Here, *getRequests()* is a site that receives the "requests" for booking a hotel for a given date. For each request received, a hotel room is booked. The received "request" is bound to variable *req* which is then used as a parameter in the site call to *bookHotel* to make the intended computation.
*f>>g* is a short form of *f>x>g* in case *g* does not use *x*. Hence,

```
bookHotel(date,req) >> bookCab(date,req)
```

would first call site *bookHotel* for a given *date* and *req* and then invoke the site *bookCab*.

**Symmetric Composition(|)**: Consider an expression $f \mid g$. Here *f* and *g* are the sites that are evaluated parallely. Values from both *f* and *g* are published. Hence,

```
(bookHotel(date,req) | bookCab(date,req)) >> Email(addr,ack)
```

would initiate the booking for a hotel room and a cab concurrently for a given date and request received. Two emails would be generated: one for booking of the hotel and the other for the cab.

**Pruning Composition(<)**: Consider an expression $f < x < g$. This is interpreted as: for some values published by *g* do *f*. Here evaluation of *f* and *g* are started in parallel. Site calls that need *x* are suspended. When *g* returns a value: (a) the value is bound to *x* (b) *g* is terminated and (c) the suspended calls are resumed. Values published by *f* are the values of $(f < x < g)$. For e.g.

```
bookFlight(date,x) <x< ( getDelta(date) | getUnited(date) )
```

invokes *bookFlight, getDelta, getUnited* services together. As and when Delta or United airline service returns a value, *bookFlight* is called. The value published by the other airline is ignored. The final value published by the expression is the one that is published by *bookFlight*.

When composing expressions, the >> operator has the highest precedence, followed by |, and then <. Orc provides certain fundamental sites for functions like *if, let, +, ...* for easy programming. A fundamental site for "," is provided to implement barrier synchronization. For example,

```
(bookHotel(),bookCab())
```

would wait for both Hotel and Cab to be booked and then only proceed further (barrier synchronization).

In Orc, the only mode for a site call is by invocation, service push is not captured in Orc.

An expression in Orc can be defined as def E($\overline{x}$) = F. This defines a function named E whose formal parameter list is $\overline{x}$ and body is expression F. A call E($\overline{p}$) is evaluated by replacing formal parameter $\overline{x}$ by the actual parameter $\overline{p}$ in the body F. Unlike a site call, a function call does not suspend if one of its arguments is a variable with no value. A function call may publish more than one value; it publishes every value published by the execution of F. Function definitions may be recursive. Orc also provides timeouts in the form of a site *Rtimer(t)* which returns a signal t after t time units.

Orc provides support for web service invocations by means of a predefined site *Web-Service(url)* which takes a parameter *url* and publishes the corresponding service client object. This service object can then be used to invoke any operation provided by the web service in the same way as any other Java object does. The interface is identical to that of any local function which makes orchestrations using Orc easy and intuitive. For e.g if Delta airlines is providing a service *getPrice()* returning the current price of any flight between two given cities and time, then the site call

```
Webservice("http://www...delta")
```

returns the service client object say *obj* which can be used to invoke the service *getPrice()* as *obj.getPrice()*.

## 3.2 SL and MSC

### 3.2.1 Temporal Logic of Safety: SL and DSL

In this section, we shall define and give a background on temporal logics of *safety* such as SL (and DSL) as given by Halbwachs et al. [Halbwachs et al., 1993].

**Definition 1**: A *transition system* $\prod$ is a quadruple $(Q, q_0, O, \lambda)$ where $Q$ is a set of states, $q_0 \in Q$ is the initial state, $O$ is a set of observables, and $\lambda$ is a ternary relation on $Q \times O \times Q$ (transition relation). A trace of $\prod$ is a sequence $(\omega_0, \omega_1, ..., \omega_n, ...)$ in $O^\infty$ (the set of finite or infinite sequences of $O$), such that there exists a sequence of states $(q_0, q_1, ..., q_n, ...)$ and

$(q_n, \omega_n, q_{n+1}) \in \lambda$ for any $n$.

**Definition 2:** A transition system $\prod$ satisfies a *safety* property $P$ (noted $\prod \models P$), iff all of its traces satisfy $P$.

**Definition 3:** Let *Prop* and *Aux* denote two disjoint alphabets of propositional symbols and auxiliary symbols respectively. A SL formula, $\psi$ (also referred to as a safety formula), is of the form $\exists x_1, x_2, ..., x_k \Box \varphi$ where $x_1, x_2, ..., x_k$ belong to *Aux*, $\Box$ is the classical "always" operator and $\varphi$ is a past formula.

**Definition 4:** Past formulas, denoted $\varphi$, have the following structure:

- Any symbol $a \in Prop \cup Aux$ is a past formula
- If $\varphi_1$ and $\varphi_2$ are past formulas, so are $\neg\varphi_1, \varphi_1 \vee \varphi_2$ and $\bullet\varphi_1$

**Definition 5:** A trace $\tau$ on a set of observables $O$ is a finite or infinite sequence of elements of $O$. A property is a mapping from a set of traces to $\{true, false\}$. $P$ is a *safety property* if and only if the following equivalence holds:
P$(\tau)$ = true $\Leftrightarrow P(\tau^{'})$ = true for any finite prefix $\tau^{'}$ of $\tau$
Satisfaction of a formula over traces on $O$ is given below.

Let $\varphi$ denote a past formula and $O = 2^{Prop \cup Aux}$. Models of past formulas are finite traces over $O$. The satisfaction of a formula by such a trace is defined as follows:
$(\omega_0, ..., \omega_n) \models a$ iff $a \in \omega_n$
$(\omega_0, ..., \omega_n) \models \neg\varphi$ iff $(\omega_0, ..., \omega_n) \not\models \varphi$
$(\omega_0, ..., \omega_n) \models \varphi_1 \vee \varphi_2$ iff either $(\omega_0, ..., \omega_n) \models \varphi_1$ or $(\omega_0, ..., \omega_n) \models \varphi_2$
$(\omega_0, ..., \omega_n) \models \bullet\varphi$ iff $n > 0$ and $(\omega_0, ..., \omega_{n-1}) \models \varphi$

The operator $\bullet$ is the classical "previous" operator of *temporal logic with past.* Its dual $\neg \bullet \neg$ is also used. The only difference between them is that $\bullet$ is $false$ on a trace of only one element, whereas $\neg \bullet \neg$ is *true*. On any longer trace, they both have the same value. Also,
$\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \neg\varphi_2)$
$\varphi_1 \supset \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2)$
$\varphi_1 \equiv \varphi_2 \triangleq (\varphi_1 \supset \varphi_2) \wedge (\varphi_2 \supset \varphi_1)$

A deterministic subset of SL, denoted DSL (Deterministic Safety Formula) is defined below:

**Definition 6:** DSL is a fragment of SL consisting of formulas of the form
$\exists x_1, x_2, ..., x_k \Box(\varphi \wedge (x_1 \equiv \varphi_1) \wedge ... \wedge (x_k \equiv \varphi_k))$ where

- $\varphi$ is a past formula
- for each i = 1 ... k, $\varphi_i$ is a past formula, where auxiliary variables may only appear under the $\bullet$ operator.

Models of safety formulas are finite or infinite traces over $2^{Prop}$. Such a trace $(\omega_0, ..., \omega_n, ...)$, $\omega_i \subseteq Prop$, satisfies an SL formula, if and only if there exists a trace $(\omega_0^{'}, ..., \omega_n^{'}, ...)$, $\omega_i^{'} \subseteq Aux$, such that $\forall n$, $((\omega_0 \cup \omega_0^{'}), (\omega_1 \cup \omega_1^{'}), ..., (\omega_n \cup \omega_n^{'})) \models \varphi$. Note that the expressive power of SL is exactly the class of regular safety properties.

### 3.2.2 Deducing Automata

The automaton A($\psi$) associated with a safety formula $\psi$ is defined on the alphabet $2^{Prop}$, and accepts a word ($\omega_0$,..., $\omega_n$...) if and only if it is a model of $\psi$. Since the behaviour of this automaton only depends on the current state and current input, the construction consists in translating a property of traces into a property of states. Each state of the automaton will correspond to a formula, and the transitions will correspond to formula rewriting, according to the scheme:

$$(\omega_0,...,\omega_n,...) \vDash \psi \Leftrightarrow \psi \xrightarrow{\omega_0} \psi_1 \text{ and } (\omega_1,...,\omega_n,...) \vDash \psi_1$$

With each past formula $\varphi$ an automaton A($\varphi$) is associated with input alphabet $2^{Prop \cup Aux}$, and whose boolean output is *true* if and only if the sequence of inputs received so far is a model of $\varphi$.

**Past Formula rewriting:** Past Formula rewriting (transformation) is a way of recording the past inputs for *future evaluations*. It is the basic mechanism for translating trace properties into state properties.

The inputs of the automaton A($\varphi$) are subsets $\omega$ of the finite alphabet $A = Prop \cup Aux$. The transitions (rewritings or transformations) are defined by means of predicate $\varphi \xrightarrow{\omega:b} \varphi^{'}$, where $\omega \in 2^A$ and b $\in$ {*true*,*false*}, which means "on the trace reduced to a singleton ($\omega$), the vaue of $\varphi$ is b."

$$\frac{\varphi \xrightarrow{\omega_0:b_0} \varphi_1 \xrightarrow{\omega_1:b_1} ... \xrightarrow{\omega_n:true} \varphi_n}{(\omega_0,....\omega_n) \vDash \varphi} \quad , \quad \frac{\varphi \xrightarrow{\omega_0:b_0} \varphi_1 \xrightarrow{\omega_1:b_1} ... \xrightarrow{\omega_n:false} \varphi_n}{(\omega_0,....\omega_n) \nvDash \varphi}$$

The transition predicate is defined by means of structural inference rules [Halbwachs et al., 1993]:

1. A formula consisting of a basic proposition always evaluates in the same way: its value is found in the input i.e
$$\frac{a \in \omega}{a \xrightarrow{\omega:true} a} , \quad \frac{a \notin \omega}{a \xrightarrow{\omega:false} a}$$

2. Boolean operators always evaluate in the same way, according to the values of their operands:
$$\frac{\varphi_1 \xrightarrow{\omega:b_1} \varphi_1^{'}, \varphi_2 \xrightarrow{\omega:b_2} \varphi_2^{'}}{\varphi_1 \vee \varphi_2 \xrightarrow{\omega:b_1 \vee b_2} \varphi_1^{'} \vee \varphi_2^{'}} , \quad \frac{\varphi \xrightarrow{\omega:b} \varphi^{'}}{\neg\varphi \xrightarrow{\omega:\neg b} \neg\varphi^{'}}$$

3. A $\bullet$ operator is always evaluated as if the current input was the first one. So it always evaluates to false. However if its operand evaluates to true, the operator is rewritten

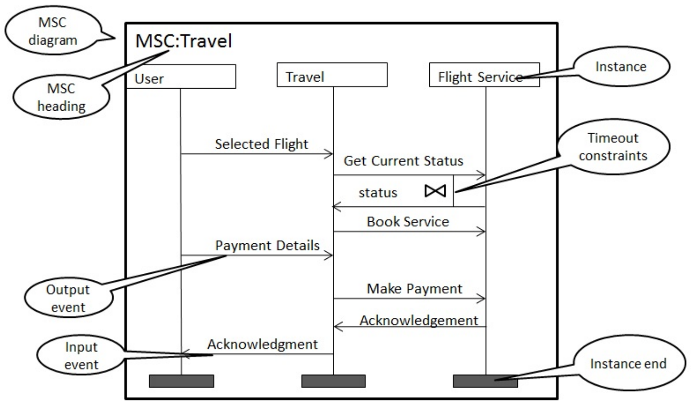


Figure 3.3: MSC representing a travel booking system

into its dual $\neg \bullet \neg$, in order to return *true* in the next state.

$$\frac{\varphi \xrightarrow{\omega:false} \varphi'}{\bullet\varphi \xrightarrow{\omega:false} \bullet\varphi'}, \quad \frac{\varphi \xrightarrow{\omega:true} \varphi'}{\bullet\varphi \xrightarrow{\omega:false} \neg\bullet\neg\varphi'}$$

4. A $\neg \bullet \neg$ operator always evaluates to true. (This is because while $\bullet$ is false on a trace of only one element, $\neg \bullet \neg$ is true.) However if its operand evaluates to false, the operator is rewritten into its dual $\bullet$, in order to return *false* in the next state.

$$\frac{\varphi \xrightarrow{\omega:true} \varphi'}{\neg\bullet\neg\varphi \xrightarrow{\omega:true} \neg\bullet\neg\varphi'}, \quad \frac{\varphi \xrightarrow{\omega:false} \varphi'}{\neg\bullet\neg\varphi \xrightarrow{\omega:true} \bullet\varphi'}$$

**Example:** The evaluation of the formula $\bullet a$ on a trace ({a},{b},{a},{b}) provides: $\bullet a \xrightarrow{\{a\}:false} \neg \bullet \neg a \xrightarrow{\{b\}:true} \bullet a \xrightarrow{\{a\}:false} \neg \bullet \neg a \xrightarrow{\{b\}:true} \bullet a$
and, since the last output is true, $(\{a\},\{b\},\{a\},\{b\}) \models \bullet a$

Let $\psi = \exists x_1, x_2, ..., x_k \Box\varphi$ where $x_1, x_2, ..., x_k$ be the safety formula. The automaton for $\psi$ can be deduced from the automaton A($\varphi$) as follows

1. All the inputs are projected onto Prop
2. Only the transitions giving the output true are retained.

### 3.2.3 Message Sequence Chart (MSC)

Message Sequence Charts (MSC) is a language to describe the interaction between a number of independent message-passing instances. A basic MSC (bMSC) structure is shown in Figure 3.3. It has instances e.g. Travel, User, Flight Service etc. with vertical lines representing system components, and horizontal arrows representing messages from one component to another. For each component, messages higher on the vertical line precede messages lower down. A timer (indicated by $\bowtie$) is used to show timeouts or delays. Figure 3.3 shows a sequence of messages against a travel agency process. On receiving a flight detail (Selected flight) from the user; the agency finds the current availability status and price using the "Get Current Status" operation. If available, it will book this flight with the "Book Service" service. It then receives the payment details from the user and proceeds for making the payment with the "Make Payment" service. On successful completion of this transaction an acknowledgement is sent to the user. Formally a MSC is defined as:

**Definition 7:** A basic message sequence chart $C$ called bMSC is a labeled directed acyclic graph with the following components:

- *Processes:* A finite set P of processes.
- *Events:* A finite set E of events that is partitioned into two sets: a set S of send events and a set R of receive events.
- *Process Labels:* A labeling function $g$ that maps each event in E to a process in P. The set of events belonging to the orchestrated process $p$ is denoted by $E_p$.
- *Send-receive Edges:* A bijection map f : S $\mapsto$ R that associates each send event $s$ with a unique receive event $f(s)$ and each receive event $r$ with a unique send event $f^{-1}(r)$.
- *Visual Order:* For process $p$ there is a local total order $<_p$ over the events $E_p$ which corresponds to the order in which the events are displayed.

The interpretation of a basic MSC in the context of runtime monitoring can be in terms of scenarios which may be wanted or unwanted. They can therefore be used to specify the monitoring properties of a business process workflow where the output/input events signify the message exchanges with the external web services.

# Chapter 4

# Realizing Workflow Patterns in Orc

The workflow patterns initiative was established with the aim of delineating the fundamental requirements that arise during business process modeling on a recurring basis and describe them in an imperative way. Aalst et al. provided a set of forty three control flow patterns which serves as a comparison mechanism for analyzing any workflow language [Russell et al., 2006]. In this chapter, an analysis of the language Orc for all forty three patterns is provided. Orc language syntax (version 1.1) is used. While the first twenty of these patterns are based on the study by Misra et al. in [Cook et al., 2006], the remaining patterns is an independent contribution of the thesis. The result is summarized with a comparison with BPMN and BPEL.

## 4.1 Workflow Patterns Support in Orc

Orc is the language of our choice for synthesizing process workflows. It is based on process calculus and has clear mathematical semantics. We provide below our analysis of the support of Orc for workflow patterns which makes it suitable for implementing business process workflows. Misra et al. [Cook et al., 2006] have provided an analysis of the initial twenty patterns in Orc. We extend this analysis for all the forty three patterns using the Orc implementation syntax (indicated with a *). The description of each pattern is quoted from Aalst et al. [Van Der Aalst et al., 2003]. Also, we have shown the implementation as per the language syntax for all patterns.

Following sites are used in our transformation:

1. **Send(m,x)**: A site that sends message $m$ to address $x$.

2. **Receive(e)**: A site that waits for a specific event $e$ and returns as soon as it occurs.

3. **Buffer: add, get(n), put, get, reset**: A site which implements a channel. The *add()* operation maintains a counter and increases every time it is called and *get(n)* operation returns as soon as this counter becomes $n$. *reset()* method resets this buffer. The *put* operation adds values to the buffer and publishes a signal on completion. The *get* operation returns an item from the buffer  it blocks until an item is available.

4. **Lock: acquire, release**: The site "lock" has exactly one owner. When the lock is created it is not owned. The method *acquire* is used to acquire the lock (i.e. becomes its owner), and all subsequent calls to acquire will block until the owner calls *release*. At that point, one of the blocked expressions, if any, will be given ownership and unblocked.

5. **Condition: set, wait**: It allows multiple activities to wait until an event happens. Before the operation *set* is called, all calls to *wait* block. When *set* is called, all waiting activities are enabled and future calls to wait return immediately.

Following definitions (Orc expressions) are used for the representation of different patterns

1. **XOR(b,f,g) $\triangleq$ if (b) >> f() else g()**: Executes $f$ if $b$ is *true*, else, executes $g$.

2. **IfCond(b,g) $\triangleq$ if (b) >> g() else signal**: Executes $g$ if $b$ is *true*, else returns a *signal*.

3. **PathNum(m1,m2) $\triangleq$**
**let(x)** $< x <$ **((receive(m1) >> let(1) | receive(m2) >> let(2) | ... )**: Waits for messages *m1*, *m2*, ..., (as defined) and as soon as any message is received, the corresponding index is published.

### 4.1.1 Basic Control-flow Patterns

1. *Sequence (WCP1)*
*Description*: "A task in a process is enabled after the completion of a preceding task in the same process."
*Implementation*: The sequential combinator ">>" is used to implement this pattern.

2. *Parallel Split(WCP2)*
*Description*: "The divergence of a branch into two or more parallel branches each of which executes concurrently."
*Implementation*: The parallel combinator "|" is used to implement this pattern.

3. *Synchronization(WCP3)*
*Description*: "The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled."
*Implementation*: This pattern is implemented by using a "," operator. If $f$, $g$ and $h$ represent the site calls and $x$ and $y$ are the values published by $f$ and $g$ respectively then the expression below represents this pattern. Here, $h$ is invoked only after both $x$ and $y$ are published.

```
(f,g) >(x,y)> h
```

4. *Exclusive Choice(WCP4)*
   *Description*: "The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches."
   *Implementation*: This pattern can be represented in Orc by using the XOR expression (defined earlier). Let $c1$ and $c2$ represent the conditions at the point of divergence. XOR is called recursively and ($!c1\&!c2$) is the translation for the default condition to ensure that at least one branch is taken. In case $h$ is not specified it could be replaced by *signal* or *stop* as the need may be.

   ```
   XOR(c1, f, XOR(c2, g, XOR(!c1&!c2,h)))
   ```

5. *Simple Merge(WCP5)*
   *Description*:"The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch."
   *Implementation*: This pattern can be represented using ">>" operator as follows

   ```
   (f|g) >> h
   ```

   Here $f$ and $g$ are executed in parallel followed by a simple merge. For each value published by $f$ and $g$, $h$ will be executed twice.

### 4.1.2 Advanced Synchronization Patterns

1. *Multiple Choice(WCP6):*
   *Description*: "The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches."
   *Implementation*: This pattern can be represented in Orc by using the IfCond expression. In this pattern more than one branches may be active and started concurrently as all the conditions are evaluated concurrently and the site given as an argument is invoked if the condition is true.

   ```
   f >> (IfCond(c1,g) | IfCond(c2,h) | IfCond(!c1&!c2,k))
   ```

2. *Structured Synchronizing Merge(WCP7):*
   *Description*: "The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The Structured Synchronizing Merge occurs in a structured context, i.e. there must be a single Multi-Choice construct earlier in the process model

with which the Structured Synchronizing Merge is associated and it must merge all of the branches emanating from the Multi-Choice."
*Implementation*: In Orc this pattern can be represented as:

```
f >> (IfCond(c1,g), IfCond(c2,h),IfCond(!c1&!c2,k)) > (x,y,z)
```

Here "," is the synchronizing operator. Since *IfCond* expression returns a *signal* when the condition is *false*, each branch in effect returns a *signal* for the synchronizing merge to proceed. It is to be noted that *c1 & c2* represents a default edge

3. *Multiple Merge(WCP8):*
   *Description*: "The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch." This pattern is similar to Simple Merge except that a simple merge is against an exclusive split and hence both incoming branches are never enabled simultaneously.
   *Implementation*: This pattern in Orc is same as that given in WCP5.

   ```
   (f|g) >> h
   ```

   Here $f$ and $g$ are executed in parallel followed by a simple merge. For each value published by $f$ and $g$, $h$ will be executed twice.

4. *Structured Discriminator(WCP9):*
   *Description*: "The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablement of incoming branches do not result in the thread of control being passed on."
   *Implementation*: In Orc it can be represented by using the *Buffer* site. Let *Discr* be an expression representing such a pattern as defined below:

   ```
    Discr() = Buffer() > S > (  (f|g|h) >x> S.add() >> stop )
                                |
                                S.get(1)
                              ) >> k
   ```

   The discriminator publishes only the first value that is placed in the buffer by $(f|g|h)$, but allows them to continue executing.

5. *Blocking Discriminator(WCP28)*:*
   *Description*: "The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The Blocking Discriminator construct resets when

all active incoming branches have been enabled once for the same process instance. Subsequent enablement of incoming branches are blocked until the Blocking Discriminator has reset."
*Implementation*: The Orc representation is similar to WCP9. It however acquires a lock at the time of divergence and releases it at the time of merge. *Buffer* site is used as before.

```
BlockDiscr() =
Lock() >L> Buffer() >S>  L.acquire >> ( (f|g|h) >x> S.add() >> stop)
                                         |
                                         S.get(1) )
                                     >> L.release >> k
```

6. *Canceling Discriminator(WCP29)*:*
   *Description*: "The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the Canceling Discriminator also cancels the execution of all of the other incoming branches and resets the construct."
   *Implementation*: Canceling discriminator is implemented similar to a Structured discriminator (WCP9) except that it uses a pruning operator $<$ as given below

   ```
   CancelDiscr() =
   Buffer() > S > ( let(x) <x< ((f|g|h) >x> S.add() >> stop)
                               |
                               S.get(1)
                  ) >> k
   ```

   As soon as $x$ is published, executing instances of $f$, $g$ and $h$ will be terminated due to the pruning operator ¡ used.

7. *Structured Partial Join(WCP30)**:
   *Description*: "The convergence of two or more branches (say m) into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Subsequent enablement of incoming branches do not result in the thread of control being passed on."
   *Implementation*: A discriminator is a special case of a partial join. Where a partial join is a $n$ out of $m$ join, a discriminator is a 1 out of $m$ join. If the *get(1)* function is replaced by a *get(n)* function, we get a Structured Partial Join.

   ```
   PartialJoin() = Buffer() > S > (  (f|g|h... >x> S.add() >> stop )
                                   |
                                   S.get(n)
                                ) >> k
   ```

8. *Blocking Partial Join(WCP31)*:*
   *Description*: "The convergence of two or more branches (say $m$) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when $n$ of the incoming branches has been enabled (where n < m)."
   *Implementation*: A blocking discriminator is a special case of a blocking partial join. Where a partial join is a $n$ out of $m$ join, a blocking discriminator is a 1 out of $m$ join. If the *get(1)* function is replaced by a *get(n)* function, we get a blocking partial join. The pattern implementation is given below:

```
BlockingPartialJoin() =
 Lock() >L> Buffer() >S>  L.acquire >> ( (f|g|h... >x> S.add() >> stop)
                                         |
                                         S.get(n) )
                                   >> L.release >> k
```

9. *Canceling Partial Join(WCP32)*:*
   *Description*: "The convergence of two or more branches (say $m$) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when $n$ of the incoming branches have been enabled where $n$ is less than $m$. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct".
   *Implementation*: A canceling discriminator is a special case of a canceling partial join. Where a canceling partial join is a $n$ out of $m$ join, a discriminator is a 1 out of $m$ join. If the *get(1)* function is replaced by a *get(n)* function, we get a canceling partial join. The pattern implementation is given below:

```
CancelPartialJoin() =
Buffer() > S > ( let(x) <x< ((f|g|h) >x> S.add() >> stop)
                              |
                              S.get(n)
               ) >> k
```

10. *Generalized AND-Join(WCP33)*:*
    *Description*: "The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings. Over time, each of the incoming branches should deliver the same number of triggers to the AND-join construct."
    *Implementation*: The Orc language semantics supports concurrent executions by default. The "," operator ensures synchronization. Since triggers need to persist between firings, one needs a channel as the Orc calculus has no way to maintain the state required without using a site. A join of expressions $f$ and $g$, that runs expression $h$ whenever results from both $f$ and $g$ arrive is given below:

```
{-
 Demultiplex two channels.
 Whenever a result is available on both a and b, take those results,
create a pair, and publish it.
-}
def demux(a,b) = (a.get(), b.get()) >pair> ( pair | demux(a,b) )

val cf = Channel()
val cg = Channel()

 f >r> cf.put(r) >> stop
| g >r> cg.put(r) >> stop
| demux(cf, cg) >(x,y)> h
```

11. *Local Synchronizing Merge(WCP37)*:*
    *Description*: "The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge."
    *Implementation*: This pattern is applicable for an unstructured flow. It is not directly supported by Orc as it is a highly structured language. Let Figure 4.1 represent a unstructured flow. A local synchronization merge at the node after activity E is provided below.

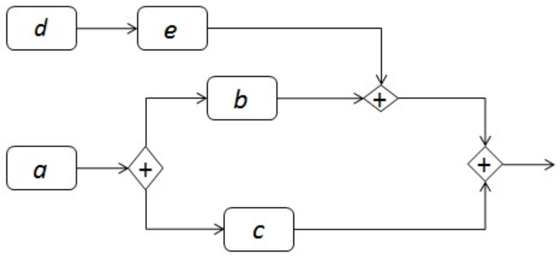


Figure 4.1: A BPMN flow for illustrating a Local Synchronizing Merge

The Orc representation is given below. The *Condition* site returns condition variable *C*. Once *e* executes *C.wait* blocks till C is set. Once *b* executes *C.set* sets this variable and the *C.wait* continues.

```
Condition >C> ( d >> e >> C.wait ) | ( a >> ( b>>C.set, c ) >> f )
```

12. *General Synchronizing Merge(WCP38)*:*
    *Description*: "The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time."
    *Implementation*: Let Figure 4.2 represent an unstructured flow with an exclusive merge. In this BPMN flow the parallel split gateway is replaced by an exclusive-or. In case the path for C is chosen then B would never be executed. For this pattern it is required that the synchronizing merge after B still proceeds. Here *C.wait* blocks till *C* is set. *C.set* is called on both the branches after *a*. On one branch i.e. for *b* it is set after the execution of *b* and for all other branches condition *C* is set as soon as the path is chosen. This allows the synchronizing merge node to proceed.

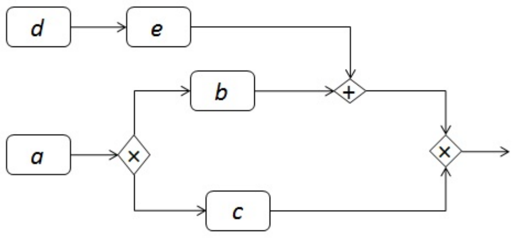


Figure 4.2: A BPMN flow for illustrating a General Synchronizing Merge

```
Condition >C> ( d >> e >> C.wait ) | ( a >> ( b >> C.set | C.set >> c ) >> f )
```

In effect for all other branches condition is set through *C.set* as soon as that path is chosen. This allows the merge node to proceed.

13. *Thread Merge(41)*:*
    *Description*: "At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution."
    *Implementation*: This pattern is implemented using *Buffer* site. For each published value of *f* (or each execution of *f*), *S.add* would increment the count. *S.get(n)* would block till this count becomes *n* and then *g* is executed.

```
Buffer >S> ( f >> S.add() >> stop
             |
             S.get(n)) >> g
```

14. *Thread Split(42)*:*
    *Description*: "At a given point in a process, a nominated number of execution threads

can be initiated in a single branch of the same process instance."
*Implementation*: Orc library has a site "each" which publishes each element in the "given" list in parallel. The value published could be in any order. The pattern is implemented by the number of threads required as a numbered list and then using the site *each* to start individual thread of execution. For e.g. if we wish to start 3 threads we can use:

```
list = [1,2,3]
{- each publishes the members of the list in any order
-}
each(list) >> f
```

### 4.1.3 Iteration Patterns

1. *Structured Loops(WCP21):*
   *Description*: "The ability to execute a task or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point."
   *Implementation*: The language Orc is highly structured and the structured loops are implemented using recursion as shown below. The site *Loop* is executed recursively. *g* is the terminating condition. If the condition evaluates to *true*, *Loop* is invoked.

   ```
   def Loop(g, f) = g >b> IfCond(b, f >> Loop(g, f))
   ```

2. *Arbitrary Cycles(WCP10):*
   *Description*: "The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches."
   *Implementation*: Workflows with arbitrary cycles and loops are created in Orc using recursive definitions like those of structured flows. This is explained by Misra et al [Cook et al., 2006] and illustrated with an example in Figure 4.3.

   Let BA represent the process shown in Figure 4.3. BA is invoked recursively if *c*3 is *true.*

   ```
   def BA = a >> XOR(c1,c,b>>IfCond(c3,BA))
   ```

3. *Recursion(WCP22):*
   *Description*: "The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated."
   *Implementation*: Recursion is supported as explained for Structured Loops (WCP21).

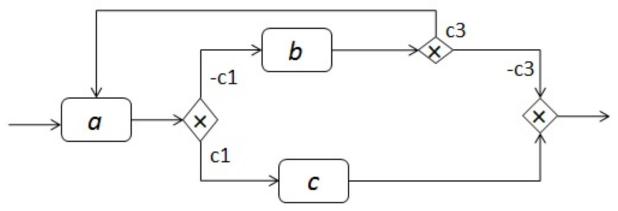


Figure 4.3: A BPMN flow for illustrating a Arbitrary Cycle

### 4.1.4 Termination Patterns

1. *Implicit Termination(WCP11):*
   *Description*: "A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successfully completed."
   *Implementation*: Implicit termination means that an expression continues running till it has to, and that no explicit *stop* action is required. Since there is no explicit stop action in Orc, it supports implicit termination.

2. *Explicit Termination(WCP43)*:*
   *Description*: "A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is canceled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed."
   *Implementation*: Orc supports implicit termination and hence no explicit termination is required. However the *stop* site can be used explicitly to terminate a particular branch of execution.

### 4.1.5 Multiple Instance(MI) Patterns

1. *MI without Synchronization(WCP12):*
   *Description*: "Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. Each of the instances of the multiple instance task that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case identifier and have access to the same data elements) and each of them must execute independently from and without reference to the task that started them."

*Implementation*: Since Orc is based on process calculus, the instantiation of instances is not differentiated as a explicit pattern to be supported. Multiple threads are created using parallel combinator and hence this pattern is supported using "|" combinator just as WCP2.

2. *MI with a Priori Design Time Knowledge(WCP13):*
*Description*: "Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered."
*Implementation*: Since the list of instances is known at design time, they can be synchronized by using "," instead of | and implemented as a WCP3 pattern

3. *MI with a Priori Run Time Knowledge(WCP14):*
*Description*: "Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered."
*Implementation*: Since the number of instances is known as a runtime quantity before the instances are created, it can be stored as a numbered list. An activity is then started by publishing each number from this list, and all the activities can then be synchronized using ",". The *Synclist* given below implements this. Here a:as represents a list in Orc where a is the first element.

```
def SyncList(F, []) = signal
def SyncList(F, a : as) = Sync(F(a), SyncList(F, as))
```

Here : is a concatenation operator in Orc. If K:L is an expression, it publishes a new list whose first element is the value of K and whose remaining elements are the list value of L.

4. *MI without a Priori Run Time Knowledge(WCP15):*
*Description*: "Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered."
*Implementation*: This pattern allows creation of instances where the number of instances is not known at design time: more instances may be created until some condition is satisfied. This pattern implementation in Orc is explained in [Cook et al., 2006] and provided below. Depending upon the condition $g$, $f$ is executed.

```
def ParLoop(g, f) = g >b> IfCond(b, Sync(f, ParLoop(g, f)))
```

5. *Static Partial join for MI(WCP34)*:*
   *Description*: "Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered. Subsequent completions of the remaining m-n instances are inconsequential, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled."
   *Implementation*: Though not very intuitive, such a workflow can be realized by using *Buffer*, *Lock* and *SyncList*. *f* would be executed *m* number of times. These *m* instances will be executed concurrently. Once *n* executions of *f* complete, *h* will execute (*S.get(n)* would publish a value). Once *n* executions of *f* complete, the lock is released signifying reset of the merge.

```
list = [1,2,...,m]
def SyncList(f, []) = signal
def SyncList(f, a : as) = Sync(f, SyncList(f, as))
Lock() >L> Buffer() >S>  L.acquire >>
                         (SyncList(f >> S.add() >> stop,list) >> L.release
                         |
                         S.get(n) ) >> h
```

6. *Static Cancelation Partial join for MI(35)**:
   *Description*: "Within a given process instance, multiple concurrent instances of a task (say $m$) can be created. The required number of instances is known when the first task instance commences. Once $n$ of the task instances have completed (where $n$ is less than $m$), the next task in the process is triggered and the remaining $m - n$ instances are canceled."
   *Implementation*: The count of $m$ executions is stored as a list. For each execution the count maintained with the *Buffer* is incremented using *add* operation and once this count becomes $n$, *get(n)* publishes a value and $h$ executes. The executing instances of $f$ are terminated.

```
list = [1,2,...,m]
Buffer() > S > (  let(x) <x< (each(list) >> f >> S.add() >> stop)
                             |
                             (S.get(n) )
               )>> H
```

### 4.1.6 State-Based patterns

1. *Deferred Choice(WCP16):*
   *Description*: "A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches

represent possible future courses of execution. The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn."
*Implementation*: The pattern is implemented using *PathNum* site which returns the index of the message received (explained in Section 4.1). Depending on the index value, either of $h$, $k$ or $l$ is executed.

```
g >> PathNum(m1:m2) >x> ( IfCond(x=1,h) | IfCond(x=2,k) | IfCond(x=3,l) )
```

2. *Critical Section(WCP39)*:*
*Description*: "Two or more connected subgraphs of a process model are identified as "critical sections". At runtime for a given process instance, only tasks in one of these "critical sections" can be active at any given time. Once execution of the tasks in one "critical section" commences, it must complete before another "critical section" can commence."
*Implementation*: Critical Sections can be implemented using locks. If CS1 and CS2 represent the two critical sections, then the lock should be acquired before CS1/CS2 commences and released later. The one which acquires the lock ensures that the tasks in the other section will not execute till it is released.

```
L.acquire > CS1 > L.release
L.acquire > CS2 > L.release
```

3. *Interleaved Parallel Routing(WCP17):*
*Description*: "A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can by completed in any order that accords with the partial order. However, as an additional requirement, no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time)."
*Implementation*: $f1...fn$ are executed concurrently. However each would first need to acquire the lock and then release it when it completes. This ensures that only one function is executed at a given time.

```
wait(M, f) = M.acquire >> f >x> M.release >>let(x)
Lock >M> (wait(M, f1) | ... | wait(M, fn))
```

4. *Interleaved Routing(WCP40)*:*
*Description*: "Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated."
*Implementation*: Very similar to the WCP17 here $f1...fn$ are started concurrently with a lock but they are all synchronized such that the next task will start only after they all complete.

```
wait(M, f) = M.acquire >> f >x> M.release >>let(x)
Lock >M> (wait(M, f1) , ... , wait(M, fn))
```

5. *Milestone(WCP18):*
   *Description*: "A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger."
   *Implementation*: The pattern execution is explained in [Cook et al., 2006]. Consider three Orc activities $f$, $g$, and $e$. The completion of activity $f$ enables $g$. Let $e$ be an event that is raised when $g$ is no longer allowed to run. Thus $f$ precedes $g$ and $e$, while $e$ can interrupt $g$.

   ```
   First(g,e)= let(x) <x< (g | e)
   Interrupt(g, e) = First(g,e)
   Milestone(f, g, e) = f >> Interrupt(g, e)
   ```

### 4.1.7 Cancelation Patterns

1. *Cancel Activity(19)*:
   *Description*: "An enabled task is withdrawn prior to it commencing execution. If the task has started, it is disabled and, where possible, the currently running instance is halted and removed."
   *Implementation*: Canceling can apply to an activity that is part of a workflow or an entire workflow case. The *Interrupt* expression as used in the previous pattern can be applied to a part of a workflow or the entire workflow to cancel part or all of the activity.

2. *Cancel Case(20)*:
   *Description*: "A complete process instance is removed. This includes currently executing tasks, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully."
   *Implementation*: Explained in WCP19.

3. *Cancel Region(25)**:
   *Description*: "The ability to disable a set of tasks in a process instance. If any of the tasks are already executing (or are currently enabled), then they are withdrawn. The tasks need not be a connected subset of the overall process model."
   *Implementation*: Here *receive(event)* is a blocking call which waits till *event* is received. It is started along with $g$ and $h$. If *event* occurs then *false* is published and the executing instances of $g$ and $h$ are terminated. The implementation of this pattern is as given below.

```
let(x) <x< ( g | h | receive(event) > let(false) )
```

4. *Cancel MI Activity(26)*:*
   *Description*: "Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. At any time, the multiple instance task can be cancelled and any instances which have not completed are withdrawn. Task instances that have already completed are unaffected."
   *Implementation*: The implementation is similar to the WCP25 except that in this multiple instances of the same process $f$ are being created in place of processes $g$ and $h$.

```
let(x) <x< ( each(list) >> f  | receive(event) > let(false) )
```

5. *Complete MI Activity(27)*:*
   *Description*: "Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered. During the course of execution, it is possible that the task needs to be forcibly completed such that any remaining instances are withdrawn and the thread of control is passed to subsequent tasks."
   *Implementation*: If $m$ instances of $f$ are created then the *SyncList* function creates these and then synchronizes them. If *event* is received before all instances of $f$ complete, then the process continues and the remaining instances of $f$ are terminated due to the pruning operator $<$ used.

```
list = [1,2,...,m]
def SyncList(f, []) = signal
def SyncList(f, a : as) = Sync(f, SyncList(f, as))
let(x) <x< ( SyncList(f ,list) | receive(event) > let(false) )
```

### 4.1.8 Trigger Patterns

1. *Transient Trigger(WCP23)*:*
   *Description:*"The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received."
   *Implementation*: A trigger can only be utilized if there is a task instance waiting for it at the time it is received. If *A1* is the activity to be triggered by an event *e1* then a transient trigger can be represented as below. After *A0*, the process flow waits for $S$ to be set (*S.wait*). On occurrence of event *e1* this condition is set. It then waits for t time-units after which it is reset (*S.reset*).

```
Condition >S> receive(e1) >> S.set >> Rtimer(t) >> S.reset
A0>>S.wait >> A1
```

2. *Persistent Trigger(WCP24)*:*
   *Description*: "The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task."
   *Implementation*: Persistent triggers are same as transient triggers with a difference that they are retained by the workflow until they can be acted on by the receiving activity. If an activity *A1* is to be triggered by an event *e1* then a persistent trigger can be realized as given below.

   ```
   Condition >S> receive(e1)>> S.set
   A0>>S.wait >> A1
   ```

## 4.2 A Comparison of BPMN, BPEL and Orc for WCP patterns

Our study of Orc patterns support proves that though Orc is based on simple constructs it is a powerful and expressive language. It is well suited for developing business process workflows. Table 4.1 gives a comparison of Orc with BPEL/BPMN [1] in terms of workflow pattern support and also helps in the analysis of the mapping of BPMN constructs with Orc that we will describe later. BPMN analysis is based on [Wohed et al., 2006]. Our study clearly indicates that Orc language has a larger support for workflow patterns than the language BPEL.

[1]BPMN 1.0 and Oracle BPEL PM is considered.

Table 4.1: Comparison of BPMN, BPEL and Orc language support for the workflow patterns

| Pattern | BPMN | BPEL | Orc | Pattern | BPMN | BPEL | Orc |
|---|---|---|---|---|---|---|---|
| *Basic Control Flow* | | | | *Termination* | | | |
| Sequence | + | + | + | Implicit Termination | + | + | + |
| Parallel Split | + | + | + | Explicit Termination | + | - | - |
| Synchronization | + | + | + | *Multiple Instances* | | | |
| Exclusive Choice | + | + | + | MI without Synchronization | + | + | + |
| Simple Merge | + | + | + | MI with Priori Design Time Know. | + | + | + |
| *Advanced Synchronization* | | | | MI with Priori Runtime Know. | + | + | + |
| Multiple Choice | + | + | + | MI without Priori Runtime Know. | - | +/- | + |
| Str. Synchronizing Merge | + | + | + | Complete MI Activity | - | - | + |
| Multiple Merge | + | - | + | Static Partial Join for MI | +/- | - | + |
| Structured Discriminator | +/- | - | + | Canceling Partial Join for MI | +/- | - | + |
| Blocking Discriminator | +/- | - | + | Dynamic Partial Join for MI | +/- | - | + |
| Canceling Discriminator | + | - | + | *State based* | | | |
| Structured Partial Join | +/- | - | + | Deferred Choice | + | + | +/- |
| Blocking Partial Join | +/- | - | + | Critical Section | - | + | + |
| Canceling Partial Join | +/- | - | + | Interleaved Parallel Routing | - | - | + |
| Generalized AND-join | + | - | + | Interleaved Routing | +/- | - | + |
| Local Synchronizing Merge | - | + | + | Milestone | - | +/- | + |
| General Synchronizing Merge | - | - | + | *Cancelation* | | | |
| Thread Merge | + | +/- | + | Cancel Activity | + | +/- | + |
| Thread Split | + | +/- | + | Cancel Case | + | + | + |
| *Iteration* | | | | Cancel Region | +/- | +/- | + |
| Arbitrary Cycles | + | - | +/- | Cancel MI Activity | + | + | + |
| Structured Loop | + | + | + | *Trigger* | | | |
| Recursion | - | - | + | Transient Trigger | - | - | +/- |
| | | | | Persistent Trigger | + | + | + |

# Chapter 5

# A MDA based approach for Synthesizing Process Workflows

In this chapter, the tool BPMN2ORC, which transforms a BPMN model to Orc, is described. It includes the mappings of BPMN core constructs with that of Orc and the implementation algorithm along with the illustration with an example.

## 5.1 Transforming BPMN to Orc: An informal introduction

Consider the example of a simple order processing BPMN process shown in Figure 5.1 that has the following workflow:

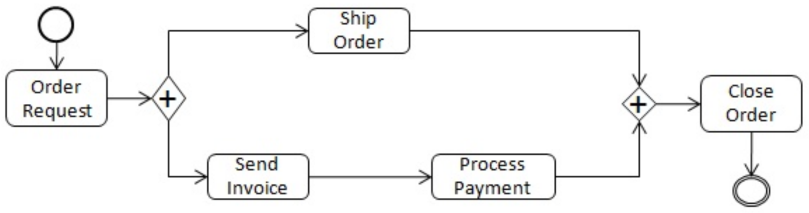


Figure 5.1: A simple order processing model

1. On receipt of an order request, concurrently start the branches with
    (a) action "Ship Order", and
    (b) "Send Invoice" action. This action is followed by the "Process Payment".
2. Wait for completion of the actions in the above two branches and then initiate the action "Close Order".

For transforming it into Orc, we need to identify the "sites" corresponding to the web services as given below:

1. *OrderRequest()*: A service which receives a PO request,
2. *Send Invoice()*: A service which sends an invoice,
3. *Process Payment()*: A service which processes payment e.g. check clearance, and
4. *ShipOrder()*: A service which initiates order shipment

An equivalent Orc workflow structure is:

```
1. OrderRequest() >>
2. (
3.     ( ShipOrder() )
4.    ,
5.     ( SendInvoice() >> ProcessPayment() )
6. )
7.    >(x,y)>  CloseOrder()
```

The initiation of the actions of the two branches of the BPMN flow after the Order Request is received is reflected in lines 3 and 5 respectively and the "," indicates the parallel branch of the workflow. The sequential action in the second branch is captured by ">>" in line 5. The parameters x and y reflect the value sent out by " Ship Order " and " Process Payment " in the first and second branches respectively; that is, the site *ShipOrder* publishes x on completion and *ProcessPayment* publishes y. Thus, "(x,y)" would wait till both x and y are available. Once x and y are both available, the action *CloseOrder* can execute; the latter operation is reflected by the sequencing operator "$> (x, y) >$" in line 7. Please note that if no value is to be published by *ShipOrder* and *ProcessPayment* the $> (x, y >$ can be replaced by a sequential combinator without any data values i.e. $>>$

The above example illustrates how the transformation of BPMN to Orc closely follows the graphical structure of BPMN into the concurrent computation graph of Orc thus, allowing the exploitation of concurrency while generating code.

We consider a core subset of BPMN objects consisting of: Activity, Gateway and Events. A BPMN diagram composed of these core entities is called a BPMN Core Diagram (BPD). The following notation is used for transformations:

1. An *Activity* of BPD is treated as a site call in Orc. A site call in Orc could be a fundamental site (function provided in Orc), a local java method, a web-service invocation or any other remote call. Activities may be also be defined as sub processes. A sub process is treated as an expression in Orc.
2. *Gateways* are implemented using the three combinators: sequential composition ($>>$), pruning ($<$), symmetric composition ($|$). For ease of programming, Orc also provides the operator (,) (implemented as a fundamental site) which is used for barrier synchronization though it is internally implemented using "$<$" as described by Misra et al. in [Misra, 2006]

3. In Orc, the only mode for a site call is by invocation, service push is not captured. Hence, *Events* are considered implemented as predefined sites in our framework and continuously poll for a particular message/event. Timers are provided in Orc as a fundamental site *RTimer*.

4. Following sites as defined in Section 4.1 are assumed.
   *Send(m,x)* , *Receive(e)*, *Buffer*, *Condition*

5. The following Orc expressions as defined in Section 4.1 are used in the translation.
   *XOR(b,F,G)*, *IfCond(b,G)* and *PathNum(m1,m2)*
   Note that, (i) *XOR* function calls F if b is *true*, else, function G is called. (ii) *IfCond* function executes G if b is *true*, else returns a signal. and (iii) *PathNum* function would wait for the messages *m1*, *m2* etc and as soon as a message is received, it returns the corresponding index.

With the above mechanisms, we now describe our transformation through the following steps:

1. Equivalent Orc fragments for core BPMN elements, and

2. Transformation for mapping BPMN flow into an equivalent Orc program.

Step (1) is given below followed by (2) in Section 5.3

## 5.2 Equivalent Orc structures for core BPMN elements

In Table 5.1, we describe the core BPMN elements and their mappings in Orc that preserves the semantics of the underlying workflow pattern expressed by the core elements.

## 5.3 Algorithm: Transforming BPMN to Orc

We now describe how transformations given in Table 5.1 can be automated. BPMN is a graph orientated language and based on entirely different meta-model than that of Orc. The transformation between them is thus not straightforward. Our transformation approach is inspired by the approach taken by Aalst et al in [Ouyang et al., 2007]. A BPMN diagram may be treated as a graph wherein:

- the nodes are essentially objects that can be further categorized as tasks or activities, events (start, end, message receipts and timer events) and gateways corresponding to the various constructs of BPMN such as $G_{ps}$, $G_{pm}$, $G_{xs}$ etc. as shown in Figure 3.2.

- the edges are the control flows between these nodes

Table 5.1: BPMN core elements and their translation in Orc

| ELEMENT | ORC TRANSLATION |
|---|---|
| Sequence flow | >>, the sequential combinator in Orc. |
| Parallel Split (Fork) (Fig.3.2(a)) | \| is the concurrency operator. The equivalent Orc translation is: `R >> (F\|G)` |
| Parallel Merge (Join) (Fig.3.2(b)) | This gateway is implemented using , operator. The Orc translation is: `(F,G) >(x,y)> R` where, x and y are the values published by F and G respectively. |
| Exclusive (XOR) Split (Fig.3.2(f)) | This gateway can be translated by using the XOR function as `XOR(c1, G, XOR(c2, H, XOR(!c1&!c2,I)))` Here XOR is called recursively and (!c1 & !c2) is the translation for the default condition. |
| Exclusive (XOR) merge (Fig.3.2(g)) | This gateway is translated in Orc as `(F\|G) >> R` In case the diagram represents a parallel split and an exclusive join, then for each activation of the incoming edge the exclusive merge would be activated again. This will result in the execution of R twice. |
| Inclusive /Complex split (Fig.3.2(d,h)) | The Orc translation of this gateway is: `F >> (IfCond(c1,G) \| IfCond(c2,H) \| IfCond(!c1&!c2,I))` where *IfCond* is as defined in Section 4.1. |
| Inclusive join (Fig.3.2(e)) | In Orc this can be translated as: `F >> (IfCond(c1,G), IfCond(c2,H), IfCond(!c1&!c2,I)) > (x,y,z)` |
| Event based split (Fig.3.2(c)) | The Orc translation uses the expression *PathNum* defined earlier which returns the index of the event that occurs first. `G >> PathNum(m1:m2) >x> ( IfCond(x=1,H) \| IfCond(x=2,I) \| IfCond(x=3,K) )` Here G,H and I are synchrnized before proceeding further i.e. an inclusive join waits for all branches to complete which is ensured in the IfCond expression. |
| Complex Merge (Fig.3.2(i)) | It Orc it can be translated by making use of the *Buffer* site. `Buffer() > S > ((G\|H\|I) > x > S.add() ) \| S.get(n)` |
| Activity Looping | Activity looping in Orc is implemented by means of tail recursion. Let g be the terminating condition expression, the Orc translation of Activity Looping is : `Loop(g,F) = g >b> IfCond(b,F>>Loop(g,F)` |

Similarly, an OrcGraph may be treated as a graph wherein:

- the nodes are essentially objects that can be further categorized as tasks (site calls), events (start, end, message receipts and timer) and gateways corresponding to the following (a) parallel combinator or "|" ($G_{par}$) (b) pruning combinator or "<" ($G_{where}$) (c) synchronizing split i.e. "," ($G_{sync}$) (d) function $XOR$ ($G_{xor}$) and (e) merge denoted as "–" ($G_{merge}$)
- the edges are the control flows between these nodes

### 5.3.1 Definitions: BPD, OrcGraph and OrcSubgraph

Formal definitions of BPD, OrcGraph and OrcSubgraph are given below:
**Definition 1:** A BPD is a tuple BPD = *(O, T, E, G, $E_s$, $E_e$, $E_m$, $E_t$, $G_{ps}$, $G_{pm}$, $G_{xs}$, $G_{xm}$, $G_{is}$, $G_{im}$, $G_{cs}$, $G_{cm}$, $G_{ds}$, F, Cond)* where:

- $O$ is a set of objects which can be partitioned into disjoint sets of tasks T, events E and gateways G,
- T is the set of tasks which can be atomic or sub processes,
- E is the set of events which can be partitioned into disjoint sets of start events ($E_s$), end events ($E_t$), events of message receipt ($E_m$) and events indicating a timer ($E_t$),
- G is the set of gateways which can be partitioned into disjoint sets of parallel fork gateways ($G_{ps}$), parallel join gateways ($G_{pm}$), exclusive-or gateway ($G_{es}$), exclusive-or merge gateway ($G_{em}$), event-based gateway ($G_{ds}$), inclusive-or gateway ($G_{is}$), inclusive-or merge gateway ($G_{im}$), complex gateway ($G_{cs}$), complex merge gateway ($G_{cm}$),
- $F \subseteq O \times O$ is the control flow relation,
- $Cond : F \rightarrow C$ is a function mapping sequence flows to conditions where $dom(F) = F \cap ((G_{xs} \cup G_{is}) \times O)$

**Definition 2:** A BPD satisfying the following properties is said to be a "well formed" BPD.

- there can be only one start event in the BPD i.e $E_s = \{s\}$,
- start event has an *indegree* of zero and *outdegree* of one. i.e. $\forall s \in E_s, in(s) = \varnothing \vee |out(s)| = 1$,
- end events have an *outdegree* of zero and *indegree* of one. i.e. $\forall s \in E_e, |in(s)| = 1 \wedge out(s) = \varnothing$,
- tasks, message events and timer events have *indegree* of one and *outdegree* of one, i.e $\forall s \in (T \cup E_m \cup E_t), |in(s)| = 1 \wedge out(s) = 1$,

- all split gateways i.e. $G_{ps}, G_{xs}, G_{is}, G_{cs}, G_{ds}$ have an *indegree* $= 1$ and *outdegree* $> 1$
- all merge gateways i.e. $G_{pm}, G_{xm}, G_{im}, G_{cm}, G_{dm}$ have an *indegree* $> 1$ and *outdegree* $= 1$
- an exclusive and inclusive split gateway contains a default condition edge
- for each object in $(T \cup G)$ there exists a path from $E_s$ to the object

**Definition 3:** An OrcDAG is a tuple OrcDAG = *(O, T, E, G, $E_s$, $E_t$, $E_{Receive_m}$, $E_{RTimer_t}$, $G_{par}$, $G_{where}$, $G_{sync}$, $G_{xor}$, $G_{merge}$, F, Cond)* where:

- $O$ is a set of objects which can be partitioned into disjoint sets of tasks T, events E and gateways G,
- T are the set of tasks which can be site calls or expression calls,
- E is the set of events which can be partitioned into disjoint sets of start events ($E_s$), end events ($E_t$), message receipt events $E_{Receive_m}$ and timers $E_{RTimer_t}$,
- G is the set of gateways (combinators in Orc) which can be partitioned into disjoint sets of parallel combinator ($G_{par}$), pruning combinator ($G_{where}$), a synchronization operator ($G_{sync}$), exclusive-or split gateway ($G_{xor}$), and merge combinators $G_{merge}$)
- $F \subseteq O \times O$ is the control flow relation,
- $Cond : F - > C$ is a function mapping sequence flows to conditions where $dom(F) = F \cap (G_{xor} \times O)$
- No cycles exist, i.e. $\forall s \in G$ where $|in(s)| > 1$, $(u \times s) \subseteq F$, $u$ should not be reachable from $s$

**Definition 4:** An OrcDAG satisfying the following properties is said to be a well formed BPD.

- there can be only one start event i.e $E_s = \{s\}$,
- start event has an *indegree* of zero and *outdegree* of one. i.e. $\forall s \in E_s$, in(s) $= \varnothing \vee |out(s)| = 1$,
- end events have an *outdegree* of zero and *indegree* of one. i.e. $\forall s \in E_e, |in(s)| = 1 \wedge out(s) = \varnothing$,
- tasks, message events and timer events have *indegree* of one and *outdegree* of one, i.e $\forall s \in (T \cup E_m \cup E_t), |in(s)| = 1 \wedge out(s) = 1$,
- all split gateways i.e. $G_{par}, G_{where}, G_{sync}, G_{xor}$ have an *indegree* of one and *outdegree* $> 1$

- all merge gateways i.e. $G_{merge}$ have an *indegree* $> 1$ and *outdegree* $= 1$
- an exclusive and inclusive split gateway contains a default condition edge
- for each object in $(T \cup G)$ there exists a path from $E_s$ to the object

**Definition 5:** An OrcSubgraph is a tuple *OrcSub* = *(O, T, E, G, $E_s$, $E_t$, $E_{Receive_m}$, $E_{RTimer_t}$, $G_{par}$, $G_{where}$, $G_{sync}$, $G_{xor}$, $G_{merge}$, F, Cond)* where:

- $O$ is a set of objects which can be partitioned into disjoint sets of tasks T, events E and gateways G,
- T are the set of tasks which can be site calls or expression calls.
- E is the set of events which can be partitioned into disjoint sets of message receipt events ($E_{Receive_m}$) and timers ($E_{RTimer_t}$),
- G is the set of combinators which can be partitioned into disjoint sets of parallel combinator ($G_{par}$), pruning combinator ($G_{where}$), a synchronization operator ($G_{sync}$), exclusive-or split gateway ($G_{xor}$), and merge combinators $G_{merge}$)
- $F \subseteq O \times O$ is the control flow equation,
- $Cond : F- > C$ is a function mapping sequence flows to conditions where $dom(F)$ $= F \cap (G_{xor} \times O)$
- if there is a cycle it must be at the start vertex. $\forall s \in E_s \ where \ |in(s)| > 1, (u \times s) \subseteq F$, if there exists a path such that $u$ is reachable from $s$ then $u \in E_s$ i.e. then $u$ is the start vertex

Using the above formal definition, out broad transformational approach is described in the following:

## 5.3.2 BPD to Orc

The transformation to Orc is achieved through the following steps assuming that the given BPD is well-formed:

Step1: The BPD is traversed to detect cycles. These cycles are extracted from the main BPD and stored as subgraphs to represent functions/expressions in Orc.

Step2: The BPD is traversed to detect any quasi-structured flows. If so, these are converted into well structured flows.

Step3: The BPD is traversed to detect any arbitrary(unstructured) flows.

Step4: The BPD and all the Subgraphs are now transformed into *OrcDAG* and *OrcSub-graphs.* While *OrcDAG* is the transformation of the main BPD the *OrcSubgraphs* represents the sub-processes e.g. the cycles detected during the transformation.

Step5: The Orc code is generated by traversing these *OrcDAG* and *OrcSubgraphs*.

These steps are described and illustrated below:

Step1: BPD diagram is analysed to check whether there are any cycles. A directed graph is acyclic iff it has no (nontrivial) strongly connected subgraphs. Each strongly connected component in the BPD is contracted to a single vertex, the resulting graph is a directed acyclic graph often called the condensed graph. The strongly connected components (subgraphs) are extracted from the main BPD and stored separately as OrcSubgraph. This transformation is depicted in Figure 5.2 where a cycle $G \rightarrow M$ is detected. It is stored as a OrcSubgraph $PG$. The main OrcDAG is collapsed to replace the cycle with a task node representing $PG$.

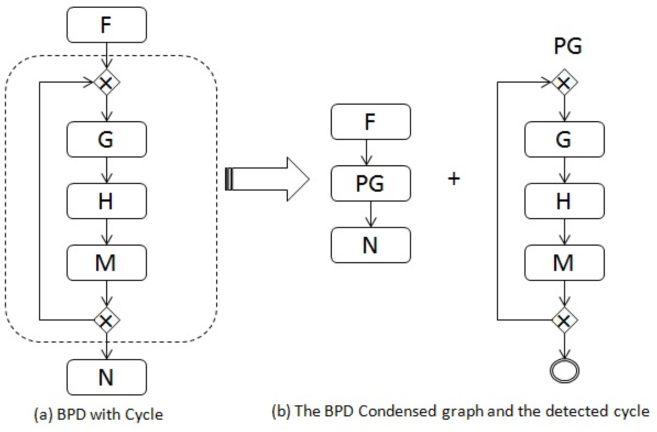


Figure 5.2: Cycles Removal from BPD

Step2: The BPD is analysed to identify certain quasi-structured components which can be translated as structured components without changing their semantics. For this, we attach a Parent Split Number (PSN) to all edges. A PSN for any edge can range from 0 to n where n denotes the number of split nodes that do not have the corresponding merge node but a higher hierarchy than the given edge. It is computed by incrementing the PSN on encountering a split node and decrementing it on the merge node. A merge for any well structured graph would have the same PSN for all its incoming edges at any merge node. In case the incoming edges have different PSN's then the graph is modified by introducing another merge node as illustrated in Figure 5.3 All edges belonging to the higher PSN are first merged. The outgoing edge of this merge node and the remaining nodes are merged to form a new node. However, it is to be noted that the number of edges with a larger PSN must be greater than the number of edges with a smaller PSN.

Step3: BPMN supports arbitrary flows. An arbitrary flow in the graph can be detected by checking the PSN numbers. If the merge nodes have different PSN on their incoming

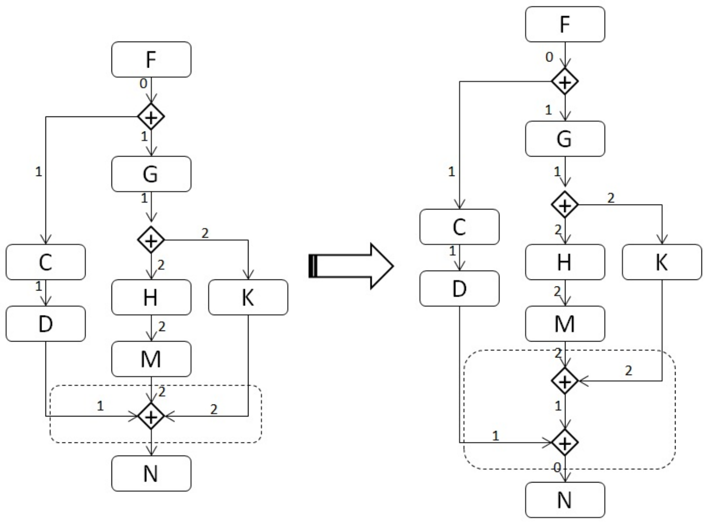


Figure 5.3: Translation of Quasi Structured Components in BPD

edges such that the number of edges with a smaller PSN is equal to or greater than the number of edges having a larger PSN, then it is an unstructured flow. An unstructured flow can be captured in Orc using Semaphores [Cook et al., 2006]. In order to generate a code that would allow such a translation the graph with an unstructured flow is converted as shown in the Figure 5.4. In steps,

- The split and merge node are converted as activity node with activity name *M.set* and *M.wait* where M is a *Condition* variable with set and wait functions
- The connecting edge is removed
- The PSN number of remaining nodes are re-calculated

Step4: The BPD and all the subgraphs are converted into OrcDAG and OrcSubgraphs as follows: (illustrated in Figure 5.5)

- All nodes representing a parallel split gateway are initially taken as a node of type | i.e $\forall u \in G_{ps}, u.type = G_{par}$
- For all parallel merge nodes we first identify the parent split node. The parent split node of these nodes are converted as of type ","(barrier synchronization) whereas the merge node remains a merge node. i.e. $\forall u \in G_{ps}, v \in G_{pm}$ where u is the parent split node of v; u.type = $G_{sync}$ and v.type = $G_{merge}$
- All exclusive split nodes are converted as $XOR$ nodes in OrcDAG i.e. $\forall u \in G_{xs}, u.type = G_{xor}$
- All exclusive join nodes are converted as $-$ nodes in OrcDAG i.e. $\forall u \in G_{xm}, u.type = G_{merge}$

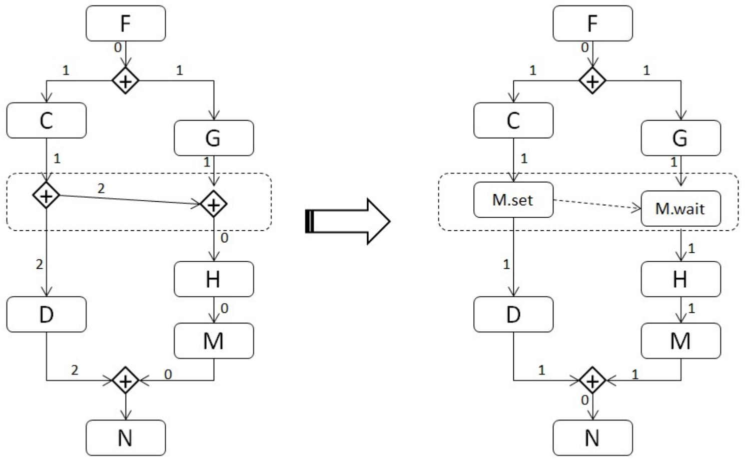


Figure 5.4: Unstructured Components in BPD and their translation

- All inclusive split nodes are converted as | nodes. i.e. $\forall u \in G_{is}, u.type = G_{par}$. Additionally the respective conditions are stored as outgoing edge names.
- For an event based gateway, insert a activity node for *PathNum* with all conditions as parameters. The gateway itself is replaced by a *IfCond* activity with conditions as indexes.
- For a complex merge, insert an activity node just after the parent split node for printing *Buffer()* $>$ *S* $>$ *(* and one just before the merge with $>$ *x* $>$ *S.add()* | *S.get()*

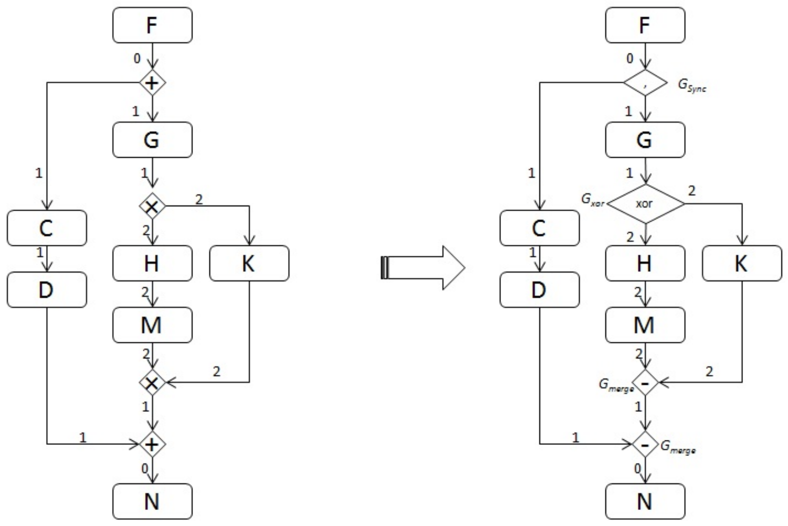


Figure 5.5: BPD translation to Orc

Step5: Orc Code is generated by traversing the OrcDAG and the OrcSubgraphs. The traversal is DFS till a merge node is encountered. It is implemented using a recursive function *Traverse(node)* which receives a node as a parameter. A node corresponds to a task, event or a gateway in the OrcGraph. A node has a name and type associated with it. Transformations corresponding to different gateways (related to Orc constructs as highlighted) and tasks are described below:

```
1. $G_{par}$:
   for all out-edges do
       if exists(out-edge.condition)
           Write IfCond (out-edge.condition, Traverse (out-edge))
       else Traverse(out-edge)
       if (out-edge <> last out-edge)
           Write '|'

3. $G_{Sync}$:
   Write '('
   for all out-edges do
       Traverse(out-edge)
       if out-edge <> last-edge
           Write ','
       else Write ')'

4. $G_{xor}$:
   for all out-edges
       if the out-edge.condition = 'Default'
           Generate the condition as a negation of all other conditions
       Write XOR (out-edge.condition
       if out-edge <> last out-edge
           Write ', lambda()=(' and then call Traverse(out-edge)
    else Write ')'

5. $G_{merge}$:
   if incoming edge = last traversed incoming edge
       If parent split node = $G_{sync}$
           Write ')'
           else If parent is $G_{xor}$ then Write 'signal' else Write >>
       Traverse(out-edge)

7. $T$:
   Write node.name + >>
   Traverse(out-edge)
```

Furthermore, due to graph based approach, certain deadlock patterns and improper looping like the ones shown in Figure 5.6 can be detected at any early stage of development. If a loop decision is made within the implicit boundaries of a set of parallel paths, then the behavior of the loop becomes ambiguous. Here (Figure 5.6(c)), it is unclear whether activity F was intended to be repeated based on the loop. Following checks are added:

1. The parent split node of a parallel merge cannot be an exclusive split.
2. The PSN of the nodes connected to the outgoing edges of the exclusive split should be greater than or equal to the PSN of its parent split node.

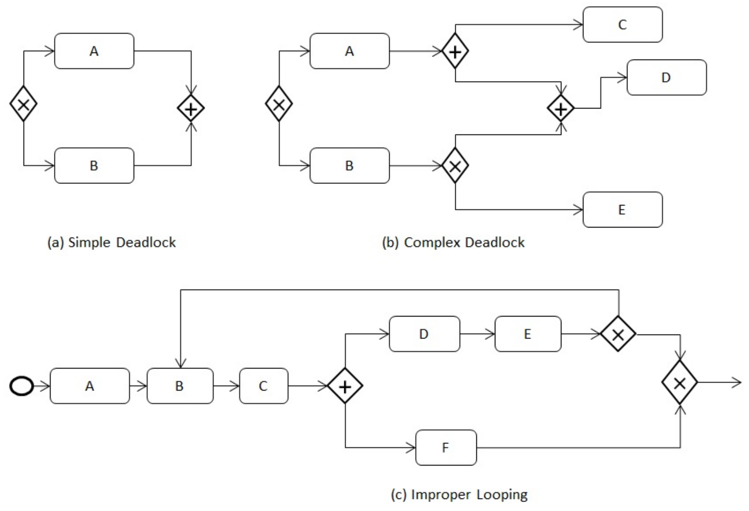


Figure 5.6: Deadlocks Detection

## 5.4 Illustration using a Order Processing System

In this section, we illustrate our BPMN to Orc transformation algorithm using the order processing system. Once an order request is received, depending on whether it is to be accepted or rejected (XOR) the order is either closed or processed. In order to process the order the shipment and the payment related tasks are initiated in parallel. Once the order shipment is complete and payment received the order can be closed. The BPMN graph is shown in Figure 5.7. We will describe the steps for the Orc code generation taking this example.

1 No cycles detected.

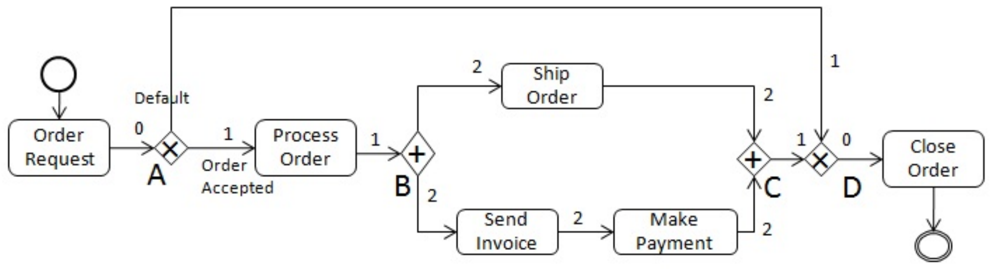


Figure 5.7: Order Processing BPD

2 All incoming edges of all merge nodes have the same PSN and hence there are no quasi structured flows.

3 The BPD is structured.

4 Considering the four gateways A,B,C,D in our example in Figure 5.7, the translation w.r.t the gateways are given below:

- All nodes of type $G_{ps}$ in BPD is translated as of type $G_{par}$ i.e | in OrcGraph. Therefore, B.type = $G_{par}$
- For all nodes of type $G_{pm}$ we first identify the parent split node. The parent split node of these nodes are converted of type $G_{sync}$ i.e. ,. The merge node remains a merge node. Therefore, B.type = $G_{sync}$ and C.type = $G_{merge}$
- All nodes of type $G_{xs}$(exclusive split) in BPD are translated as of type $G_{xor}$ in OrcDAG Therefore, A.type = $G_{xor}$
- All nodes of type $G_{xm}$(exclusive merge) in BPD are translated as of type $G_{merge}$ in OrcDAG Therefore, D.type = $G_{merge}$

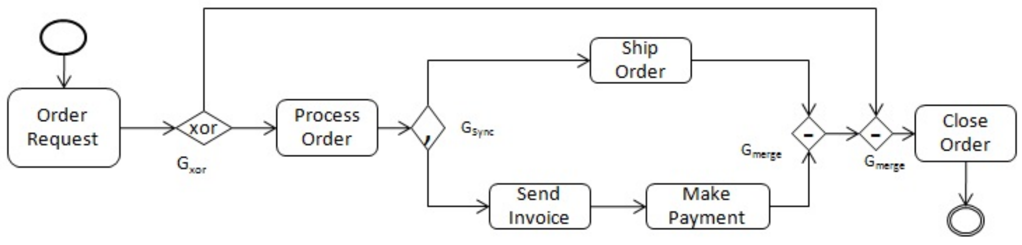


Figure 5.8: Order Processing OrcGraph

5 In this step the OrcGraph is traversed as *Traverse(ProcessOrder)* where *ProcessOrder* is the start vertex. We provide a rough sketch of the sequence of Traverse function calls and the corresponding output.

(a) Since the node.type = $T$ it outputs **OrderRequest()** >> and then calls *Traverse (XOR)*

(b) Since node.type = $G_{xor}$ it outputs **XOR(OrderAccept,(lambda() = (** and then calls *Traverse(ProcessOrder)*. On return it outputs **,lambda() = (** and then calls *Traverse($G_{xor}$)* again.

(c) Since node.type = $T$ it outputs **ProcessOrder()** >> and then calls *Traverse($G_{Sync}$)*

(d) Since node.type = $G_{sync}$ it outputs **(** and then calls *Traverse(ShipOrder)*. On return it outputs **,** and then calls *Traverse(SendInvoice)*. This is followed by the output **)**. Similarly, *Traversal(ShipOrder)* and *Traversal(MakePayment)* are called and the output comes as **( ShipOrder(),(SendInvoice()** >> **MakePayment())**

(e) Since node.type = $G_{merge}$ (there are two merge nodes; their traversal is called sequentially) then depending upon the type of parent split node either **)** or **)** >> is output.

(f) The last node.type = $T$ and it is the end node. It generates **CloseOrder()**

```
OrderRequest() >>
    XOR( OrderAccept,
          ( Lambda() = ( ProcessOrder() >>
             ( ShipOrder(),
               (SendInvoice()>>MakePayment())
              )
           ),
           Lambda() = signal
        )
>> CloseOrder()
```

## 5.5 Implementation Technologies

The system BPMN2ORC realizes the transformations discussed in this chapter and has been implemented using Java. The BPMN models are created in Eclipse BPMN modeler. The XML representation of the BPMN model is parsed using *XML Document Builder*. The graphs have been implemented using the *JGraphT* package. For detecting cycles, we have used predefined functions available in the *JGraphT* package. The generated code is then tested with the Orc-engine ver 1.1.

# Chapter 6

# Runtime Monitoring Framework

This chapter, introduces a travel agency process and a SLA that is used for illustrating the system in the later chapters. The example is used to discuss the approach for runtime monitoring, the methodology used and the key component structure of WF_SLA_MON.

## 6.1 Application v/s Management Modeling

The strategic goals of any organization are given by its Business Level Objectives(BLOs). These BLOs are defined at the enterprize level and are always kept at the back drop in the whole exercise of any process to service decomposition in that organization. Examples of such Business Level Objectives include better customer experience, increased revenues etc. Organizations enter into Business Level Agreements(BLAs) with their partners in order to meet these objectives. These agreements are however defined by humans in an informal manner and hence cannot be measured or monitored. More formal contracts are therefore derived from these BLAs to serve as a "binding" between the partners. These are called Service Level Agreements (SLAs). The key elements of these formal agreements are Service Level Objectives(SLOs) or the monitoring properties. While SLA is the entire agreement that specifies what service is to be provided, how it is supported, times, locations, costs, performance, and responsibilities of the parties involved etc, SLOs are generally specific measurable characteristics of the SLA such as availability, throughput, response time etc. SLOs typically contain QoS metric calculations but they may also include safety properties which are based on events related to a bounded history. Figure 6.1 depicts this methodology where the application and management modeling is initiated in parallel keeping in view the high level business objectives. The application modeling includes process modeling followed by its implementation using any orchestration language like BPEL, Orc etc. The orchestrated process is composed of external web service interactions which form the backbone of monitoring engine. In management modeling, BLOs are transformed into BLAs and SLAs which are composed of SLOs. In this framework, they exist in the form of executable automata's generated from the formally defined SLOs, and receives as input events the web service invocation messages. Any violation is immediately

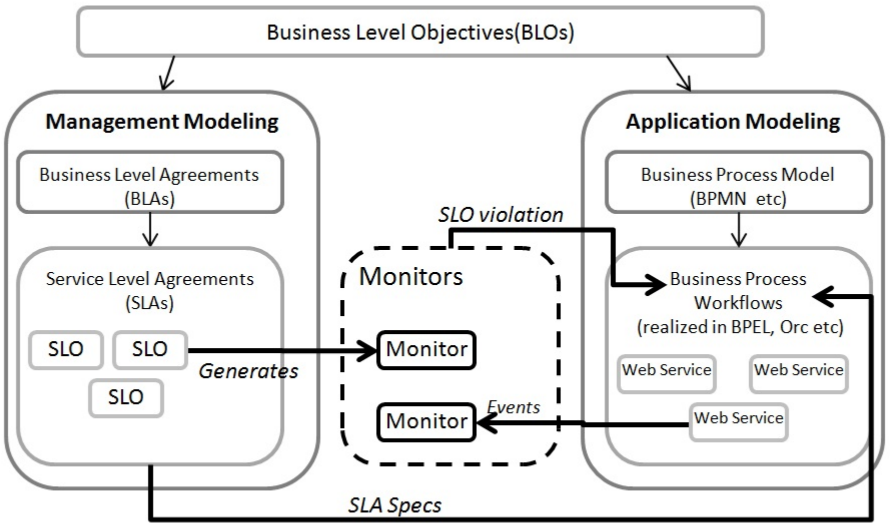

Business Level Objectives(BLOs)
Management Modeling
Business Level Agreements (BLAs)
Service Level Agreements (SLAs)
SLO
SLO
SLO
Generates
Monitors
Monitor
Monitor
Events
SLO violation
Application Modeling
Business Process Model (BPMN etc)
Business Process Workflows (realized in BPEL, Orc etc)
Web Service
Web Service
Web Service
SLA Specs

Figure 6.1: Application v/s Management Modeling

detected and communicated to the process such that it can adapt to the environment by taking an alternate or recovery step.

## 6.2 A Running example: EasyTravel

This section discusses the need for runtime monitoring and its scope using a running example of **EasyTravel** process. Consider a simple travel process named "EasyTravel" modeled as a BPMN process in Figure 6.2. It provides bookings for various airline services. The process starts when it receives a search criteria from the user in terms of price limit, date range, destination etc **(Receive Search Criteria)**. Based on the search criteria it fetches the list of services meeting the criteria **(Fetch Services)**. From the services fetched, it selects the "top services" in terms of response time and other customer ratings. These services are presented to the user for final selection. On receiving the user selection **(Receive User Selection)** it contacts the individual selected service for finding the current status and price **(Get Current Price and Status)**. If the service is available, the booking is processed **(Process Booking)**. It then waits for the user to provide payment details **(Receive Payment Details)** and parallely listens for any cancelation request **(Receive Cancelation Request**. The payment details are passed to the selected service for the actual payment **(Process Payment)**. On receiving an acknowledgement for the payment the transaction is considered as complete and an acknowledgment is sent to the user. At any time if a cancelation request is received, the booking is canceled and depending upon the user choice either the booking is tried again or the program exits. From Figure 6.2 it easily follows that the external interaction of EasyTravel is via messages as shown in the diagram in dashed lines. One can identify two service interface categories:

1. Airline Services interface providing the following web-services:
    - *GetCurrentAvailability*: Checks availability of a flight service.

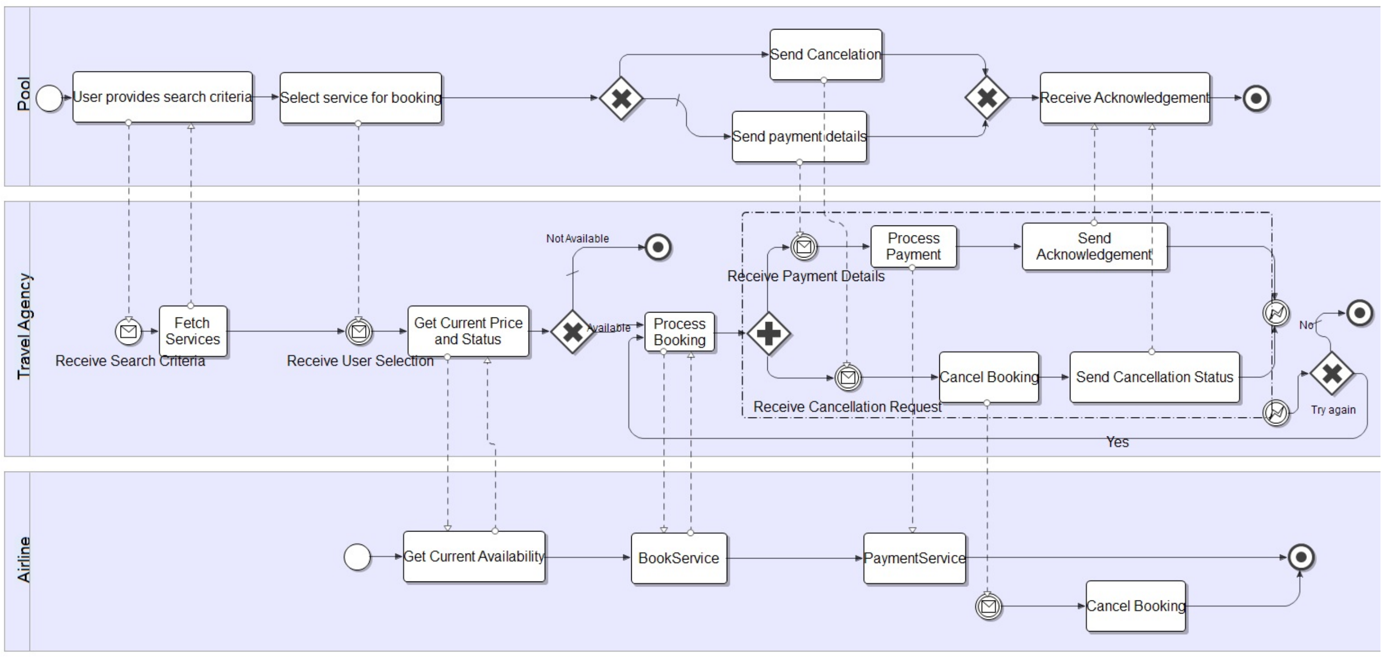


Figure 6.2: EasyTravel using BPMN

- *BookService*: Books a flight service
- *PaymentService*: Processes the payment

2. User Interface using the following web services:

- *ReceivePaymentDetails:* Receives payment from the user
- *ReceiveSearchCriteria:* Receives the search criteria from the user
- *ReceiveUserSelection:* Receives the list of services selected by the user
- *ReceiveCancelationRequest:* Receives the cancelation request from the user

## 6.3 EasyTravel's SLA

An SLA sets the expectations between the consumer and provider. It is a cornerstone of how the service provider sets and maintains commitments to service consumer. Runtime monitoring is required to ensure adherence to SLAs as they are often associated with penalty clauses. A good SLA addresses the following aspects

- What the provider is promising
- How the provider will deliver on those promises
- Who will measure delivery and how

- What happens if the provider fails to deliver as promised

EasyTravel can be viewed as a “provider” of booking service to its customers, or, as consumer of airline services. Hence, it will:

1. define and monitor SLAs for its customers
2. monitor SLAs that are given by its providers (airlines)

Let us consider an SLA for customers of EasyTravel as given below [1]:

1. Standard terms applicable to all Service Levels outlined herein:

   a Definitions

      i “Customer” refers to the organization that has purchased EasyTravel Service Bus Services.

      ii “Claim” means a claim submitted by Customer to EasyTravel pursuant to this SLA that a Service Level has not been met and that a Service Credit may be due to Customer.

      iii “Customer Support” means the services by which EasyTravel may provide assistance to Customer to resolve issues with the EasyTravel Services.

      iv “Service” or “Services” refers to the EasyTravel process for making flight bookings.

      v “Service Credit” is the percentage of the monthly service fees for the Service that is credited to Customer for a validated Claim.

      vi “Service Level” means standards EasyTravel chooses to adhere to and by which it measures the level of service it provides as specifically set forth below.

   b Service Credit Claims

      i To submit a Claim, Customer must contact Customer Support and provide notice of its intention to submit a Claim.

   c SLA Exclusions: This SLA and any applicable Service Levels do not apply to any performance or availability issues:

      i. Due to factors outside EasyTravels reasonable control:

         A. That resulted from Customer’s or third party hardware or software;

         B. To perform regular platform upgrades and patches.

   d Service Credits

      i The amount and method of calculation of Service Credits is described below in connection with each Service Level description.

      ii Service Credits are Customer’s sole and exclusive remedy for any violation of this SLA.

[1]This is adapted from the real case SLA of Microsoft Windows Azure AppFabric Service Bus

iii The Service Credits awarded in any billing month shall not, under any circumstance, exceed Customer's monthly Service fees.

2. Service Levels

a Monthly Uptime Service Level

i Definitions

A. "Downtime" is the total accumulated minutes when there is no connectivity (availability) between a customer's service endpoint and EasyTravel's gateway, as measured and aggregated in five minute intervals. A five-minute interval is marked as unavailable if all the customer's attempts to establish a connection to the Service Bus fail throughout the interval

B. "Maximum Available Minutes" is the total accumulated minutes during a billing month summed across all registered Internet facing endpoints.

C. Availability Uptime: "Monthly Uptime Percentage" for a specific Customer is the total number of "Maximum Available Minutes" less "Downtime" divided by "Maximum Available Minutes" for a billing month. It is reflected by the following formula:
*(Maximum Available Minutes - Downtime) / (Maximum Available Minutes) = Monthly Uptime Percentage*

ii Uptime Service Levels

| Monthly Uptime Percentage | Service Credit |
|---|---|
| $< 99.9\%$ | 10% |
| $< 99\%$ | 20% |

b Customer based Response time Service levels

i. "Customers" using the EasyTravel Service will be categorized as "Gold" or "Premium" customers as given below. The "Average Response Time" for bookings for customers will depend on this rating.

A. any booking made for a value 1000 INR or above increases the customer rating by 5 points

B. all usage of the service where services to be booked are searched but not actually ordered reduces the rating of the customer by 10 points

C. all cancelations made immediately after the booking in the same process instance reduces the rating by 15 points

ii. Response time Service level

c EasyTravel guarantees that under no circumstances can a payment get debited from customers account on cancelation of a booking request. In case of violations a service credit of 5% will be given against each violation.

| Cust. Rating | Cust. Type | Avg. Resp. time | Service Credit |
|---|---|---|---|
| $< 200$ | Classic | 1000ms | 5% |
| 200-400 | Premium | 500ms | 5% |
| $> 400$ | Gold | 200ms | 2% |

## 6.4 An Informal introduction to our monitoring approach

Let us assume that EasyTravel needs to ensure the satisfaction of the above SLA and assure that there are minimum service credits for its customers. The SLA described above is "informal" and cannot be checked algorithmically. Thus, the first task is to extract measurable observable properties over message interactions happening with the stakeholders of EasyTravel workflow engine. One possible approach with respect to the above SLA is given below:

1. Monthly Uptime Service level: The uptime of EasyTravel process depends on its partner airline services. Therefore one needs to ensure that "The Monthly uptime percentage for any given airline service should atleast be 99%." where "Monthly Uptime Percentage" is computed as per the definition given in the SLA. A "Downtime" is considered as a *timeout* between any web service request sent and response received. A *timeout* of 5 mins indicates that the service is unavailable. The property *p1* given below helps in satisfying this Service Level.
   *p1: A response against a BookService request should arrive with-in 5 mins.*
   Monitoring this property would help in detecting web-services which are unavailable and hence allows EasyTravel take alternate actions by replacing them with other services. Maintaining the event log of this property in the form of a data-store will also help EasyTravel in validating the service credit claims of any customer.

2. Response Time Service levels: The SLA distinguishes customers as "Classic", "Gold" and "Premium". These are based on ratings that are determined by observing their behavior over a period of time. The determination of customers ratings and the response time can be ensured as per the properties given below:

   (a) *p2: The total order value must not be more than 1000 INR".*
   In case of a higher value the rating of the customer is increased by 5 points.

   (b) *p3: Once the BookService response is received the user should be prompted for sending the payment information i.e. ReceivePaymentDetails. On receiving the payment information, actual payment i.e. PaymentService should eventually be invoked (Wanted scenario)".*
   In case of violation, the customer rating of the customer is reduced by 10 points

   (c) *p4: A BookService response is followed by PaymentService request. The response of PaymentService is immediately followed by a Cancelation request. (Unwanted*

*scenario).*
Occurrence of such a scenario reduces the customer rating by 10 points.

   (d) Other than determining the customers ratings, the SLA also requires monitoring average response time against all booking services.
   *p5: The average time taken by any airline for a booking of a classic customer must never exceed 1000 ms.*
   In case of a violation the airline would be removed from the list of available services. (RemoveAirlineFromList). Similar property is to be defined for gold and premium customers.

3. EasyTravel guarantees that under no circumstances can a payment get debited from customers account on cancelation of a booking request. This Service level is assured using safety properties as defined below:

   (a) *p6: The PaymentService must not be invoked unless the service is Booked.*

   (b) *p7: Any time PaymentService service is called, either Cancel Booking has never occurred before, or BookService has occurred since the last occurrence of Cancel Booking.*

   In case these properties are not satisfied, an SMS is generated for the administrator

4. Further *EasyTravel* maintains a list of service registries of all service providers. In order to select the "top" service for ensuring the response time promised to its customers it queries this registry database to find the average response time of various service over a period of time. The registry database is therefore updated with run-time information of QoS metrics and other profiling information related to each airline service.

5. EasyTravel may also need to distribute the booking service across many providers and it therefore maintain a history of services selected. If the last service selected is of Delta it will next select United assuming the QoS values are satisfied.

6. To choose the services based on customer rating the provider having the best QoS response is selected for higher priority customers (Gold/Premium Customers). Clearly to implement such a functionality one needs a mechanism to collect data at runtime and provide real time responses.

The monitoring of properties p1 to p7 would enable EasyTravel realize it's SLA, is argued below. Since the availability of EasyTravel depends on the external airline services on which it has no control, property p1 would allow EasyTravel to monitor availability of it's component services and use alternate service in case any service is not available. This would ensure that its monthly uptime availability is maximum. For ensuring the response time service level it needs a mechanism to:

1. identify customers as “Gold” or “Premium” which is achieved by monitoring property p2, p3 and p4 and accordingly update customer ratings

2. monitor property p5 and on receiving an “alert” remove the service from the service list temporarily

Properties p6 and p7 help EasyTravel monitor the last Service Level of the SLA. Any alert against these properties generates a SMS for the administrator to analyze the transaction and revoke payment if required. Thus, the given SLA can be satisfied by monitoring the above properties continuously as it services its stakeholders.

WF_SLA_MON, can capture these properties to monitor the given SLA. The approach for monitoring is based on “Observer based monitoring” as explained in Section 6.5. The monitor coexists with the workflow process being monitored. This is shown under the *Monitor* pool in the BPMN of Figure 6.3. The monitor, comprising many observers, continuously listens for *events* (signals) which are generated on interception of the messages against properties p1 to p7. These messages are indicated in Figure 6.3 as $*(p_i)$ representing “event” generation for monitoring a property $p_i$. The events are received by the monitor, which then verifies these properties (p1 to p7) associated with that event and generates an *alert* if the property is not satisfied. Depending on the property, the monitor may perform some computations before validating the property. The EasyTravel process, in turn, receives these *alerts* and takes associated action. Alerts are also received by the workflow administrator’s console as well as the service providers profile updater as shown in Figure 6.4.

## 6.5 An overview of WF_SLA_MON

An observer based approach of monitoring is widely used in event driven systems and synchronous frameworks. Here, an observer is a program that executes in parallel to a process and acts as a watchdog to ensure that the monitoring properties are adhered to. Let us denote the program as $P$ (the workflow) and let us say that we need to verify a safety property $\psi$ (specified using SL/bMSC). Naturally, program $P$ violates a safety property $\psi$ at a precise step (trace). Thus, we can build an observer that sends an alarm whenever the property is violated by $P$. The observer accesses the same environment as $P$. $\Omega_\psi$ is the observer of $P$ (the automata) working in parallel with $P$ as shown in Figure 6.5. In this setup, instead of proving $\psi$ about P, we see that P $||$ $\Omega_\psi$ does not emit an alarm. This scheme works when we need to consider only finite traces. For liveness properties this scheme works by monitoring whether the program terminates before the property is satisfied. The advantages of this approach are: (a) no need for explicit synchronization of $P$ and $\Omega_\psi$ (b) Modular as $P$ and $\Omega_\psi$ can evolve independently and (c) specification is executable.

The main idea lies in treating the properties to be monitored in a conjunctive manner and integrating the observers (some of them even work as actuators) with the main workflow engine. Broadly, the task of WF_SLA_MON consists of the following broad steps:

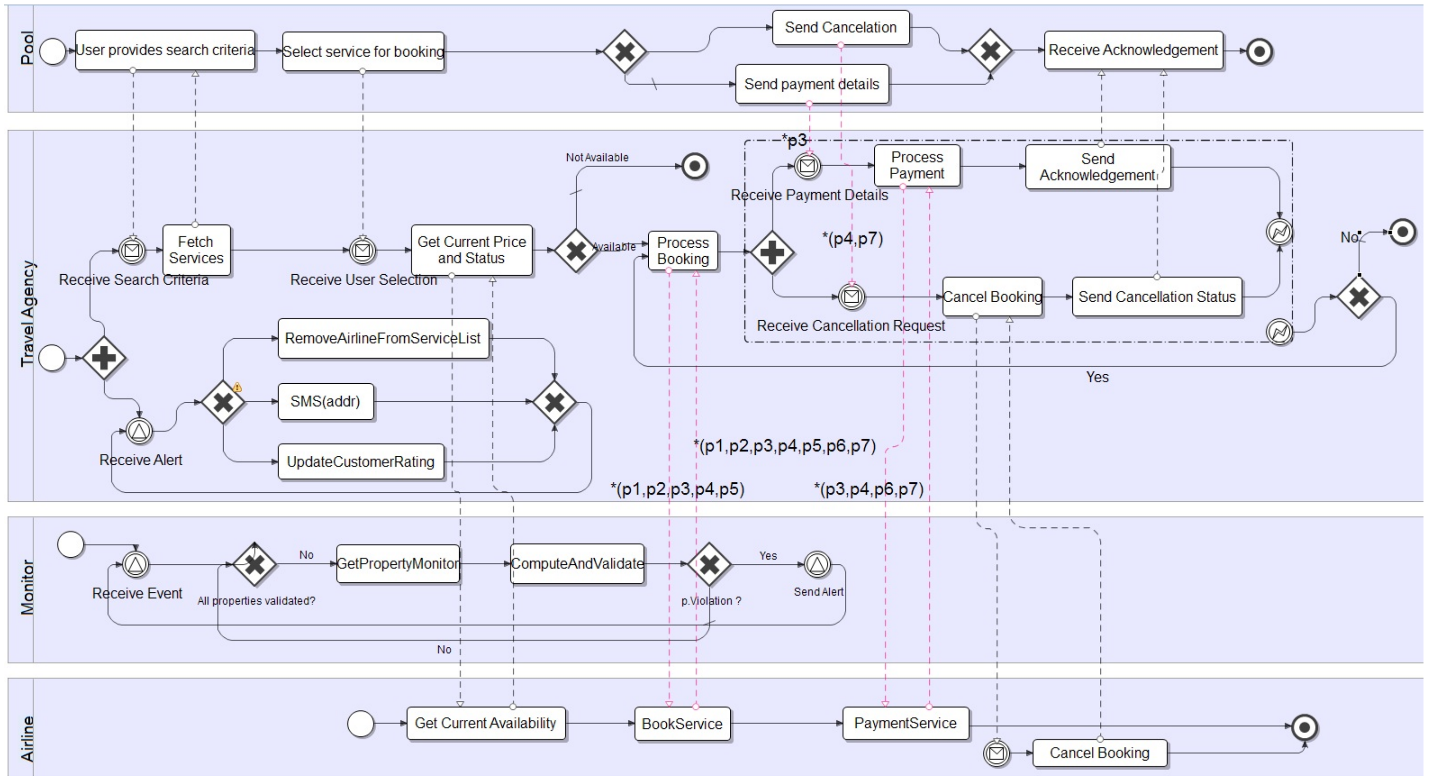


Figure 6.3: EasyTravel SLA Monitoring

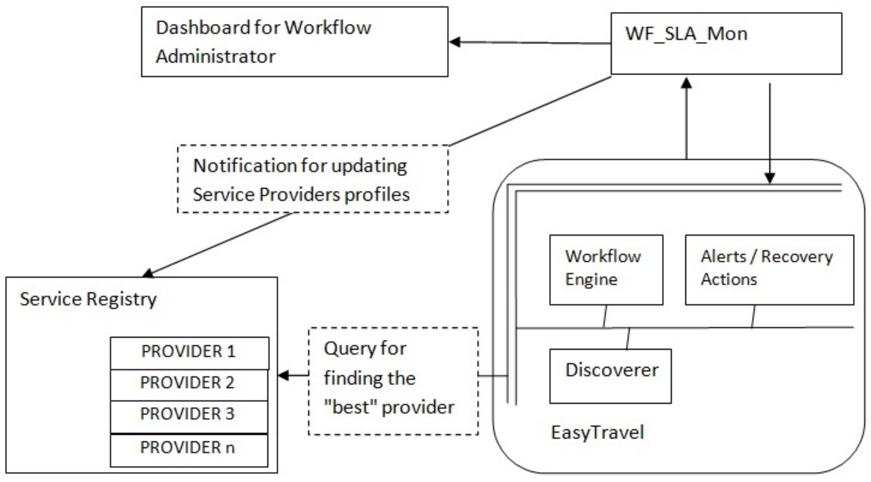

Dashboard for Workflow Administrator
WF_SLA_Mon
Notification for updating Service Providers profiles
Service Registry
PROVIDER 1
PROVIDER 2
PROVIDER 3
PROVIDER n
Query for finding the "best" provider
Workflow Engine
Alerts / Recovery Actions
Discoverer
EasyTravel

Figure 6.4: EasyTravel process's and monitor interactions

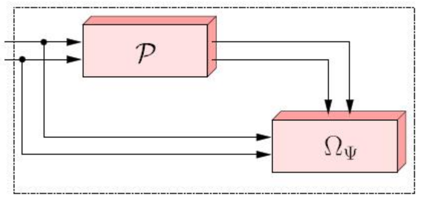


Figure 6.5: The Synchronous Observer

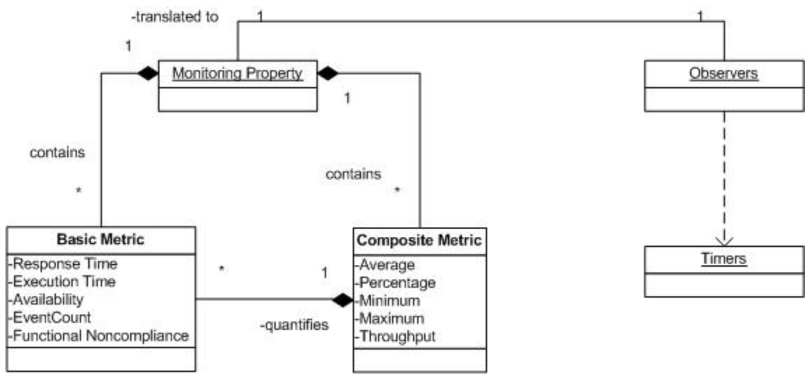


Figure 6.6: Meta-model for Monitoring Property

a Generate observers for each of the property to be monitored

b Integrate these observers with the workflow engine

c If necessary, integrate observers from non functional properties as well

As a first step, one needs to specify the properties formally. Properties are considered to be *basic* or *composite*. Basic properties comprise of QoS metrics like Response time, Execution time and Availability etc or they can represent a count of any functional non-compliance of the process. Table 6.1 gives the basic metrics provided in our system and their interpretation. Aggregation on composite metrics like *average, minimum, maximum, percentage and throughput* is also possible. A meta model for the class of monitoring properties captured in WF_SLA_MON is given in Figure 6.6. A functional noncompliance of the process can be specified in terms "safety" or "realtime response" properties SL and bMSC.

The non-functional property specification use pre-defined functions provided in our system and are specified directly using an XML. For example, the specification of properties p1 and p2 is given in Table 6.2. Here, the *timeout* tag specifies the timer constraint of 300 ms. A quantifier *Count* is applied against each timeout. If the *Count* goes above the threshold value (using a pre-determined value of x), the airline is removed from the list of services temporarily. For property p2 the message is scanned to extract the parameter *cost*

Table 6.1: Basic Metrics

| Name | Description |
|---|---|
| Response Time | It measures the current response time in milliseconds to access a web service |
| Execution Time | It measures the current execution time in milliseconds to execute a series of web service invocations as part of the process |
| Availability | It measures the current availability of a web service at all time or slots of time |
| Count | It is the count of the occurrence of any event |
| Functional Noncompliance | These are measured as the number of times a given invariant or property is violated. |

indicating the order value and the customer rating is increased if the value is more than 1000.

Functional properties are specified using SL or MSC (described in next chapter). From the specified properties observers are generated automatically in the form of an automata whose transitions correspond to the "events". These are used for "safety" properties or the wanted/unwanted scenarios requiring real time response. The generated automata is wrapped with timers and other quantification functions to form the final observer. The process communication with the external web services is in the form of SOAP messages which are intercepted by the observers. In case of any violation, an "alert" in the form of a signal is generated for the workflow engine which initiates a recovery action synchronously. This is depicted in Figure 6.7 showing the architectural diagram.

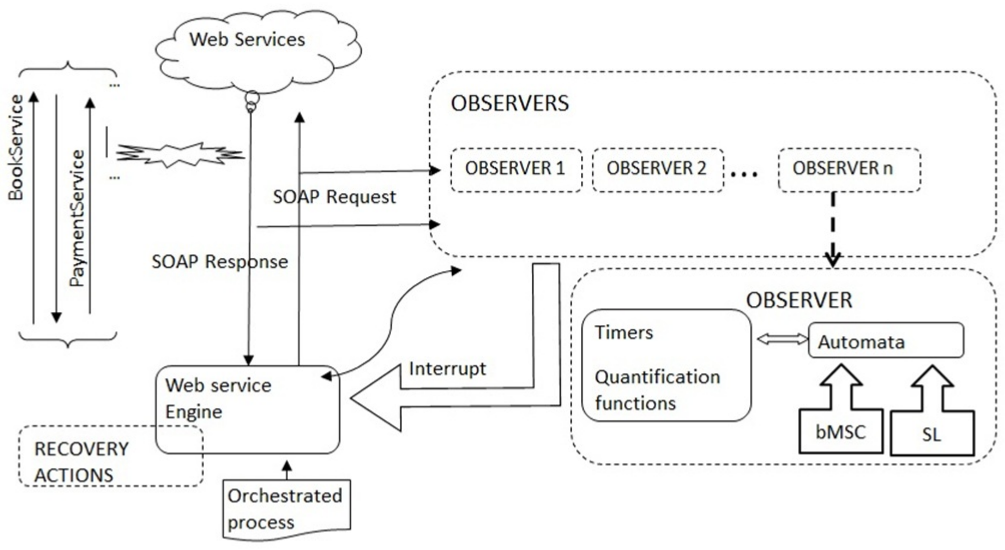


Figure 6.7: Wf_Sla_Mon Framework

Table 6.2: Property p1 and p2

| p1 | p2 |
|---|---|
| <monitor> | <monitor> |
| <name> p1 </name> | <name> p2 </name> |
| <metric> **ResponseTime**</> | <param>Cost</param> |
| <quantifier> **Count**</> | <value> **1000**</value> |
| <value> x</value> | <service> |
| <service> | <portname> **Delta**</> |
| <portname> **Delta**</> | <input-events> |
| <input-events> | <request> |
| <request> | <oper> **BookService**</> |
| <oper> **BookService**</> | </request> |
| </request> | <response> |
| <response> | <oper> **BookService**</> |
| <oper> **BookService**</> | </response> |
| </response> | </input-events> |
| </input-events> | </service> |
| </service> | <action> **IncreaseCustomerRating**</> |
| <timeout>300</timeout> | </monitor> |
| <action> **RemoveAirlineFromList**</> | |
| </monitor> | |

In Chapter 7, MSC and SL specification is described along with their transformation to automata.

# Chapter 7

# Monitor Specification and Realization

A functional noncompliance of the process (one of the basic metric) can be specified in terms of invariants or properties required to be satisfied by the process for its correctness. It can be measured as the number of times a particular safety property is violated, an unwanted scenario occurred, or any occurrence of an event depending upon the past history. Monitoring properties can be specified using Message Sequence Charts(MSC) or the language SL. This chapter provides the details of the algorithms and methodology used. The chapter covers the following topics (a) Property specification using MSC, (b) MSC to Automata transformation algorithm, (c) Property specification using SL, (d) SL to Automata transformation algorithm, and (e) Integration of generated observers with the runtime engine of workflows. These are illustrated by using properties derived from the EasyTravel SLA as given in the previous chapter.

## 7.1 Using MSC

### 7.1.1 Specification using MSC

A bMSC provides a graphical and intuitive representations of "unwanted" or "wanted" scenarios. Where an *unwanted scenario* is a sequence of message exchanges which must never occur (representing a safety property), a *wanted scenario* is the sequence of messages that are desirable. bMSC can also be used for specification of timeout criteria.

Given below are three examples of monitoring properties from the EasyTravel process and their specification in bMSC.

**Example 1**: The response time of *BookService* should not be more than x ms for Delta airlines.

**Example 2**: Payment i.e. *PaymentService* must never happen unless the response against

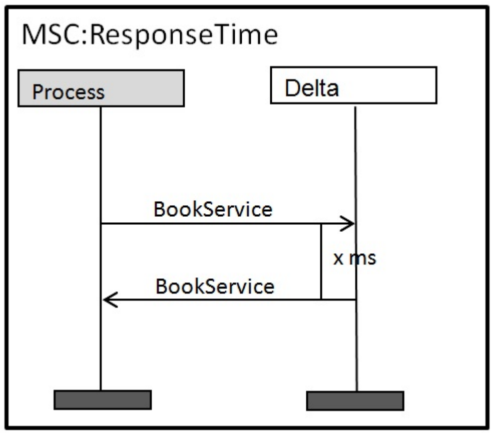


Figure 7.1: Response-time MSC (Example 1)

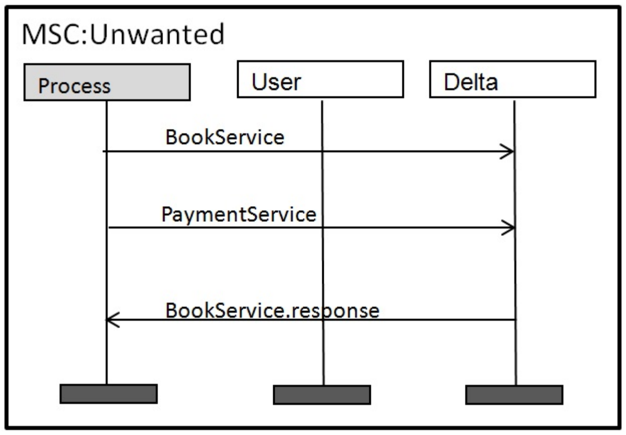


Figure 7.2: Unwanted scenario (Example 2)

*BookService* request is received. (Unwanted scenario)

**Example 3**: *BookService* request should always be followed by the *BookService* response. Only after the response is received should the user be prompted for sending the payment information. Once the payment information is received, actual payment i.e. *PaymentService* is invoked (Wanted scenario)

The three example properties are specified using MSC as shown in Figure 7.1, 7.2 and 7.3. Here, the instance "Process", is for travel, and is shown in grey to depict our travel process (which is being monitored). Any arrow going away from it is a *request* message whereas the one coming towards it is a *response* message. Figure 7.1 denotes that the response time for a *BookService* must be more than x secs for Delta airlines (*service provider*). Figure 7.2 depicts an unwanted scenario. Occurrence of such a message sequence is "unwanted" and must therefore be detected by the runtime monitor. Figure 7.3 shows

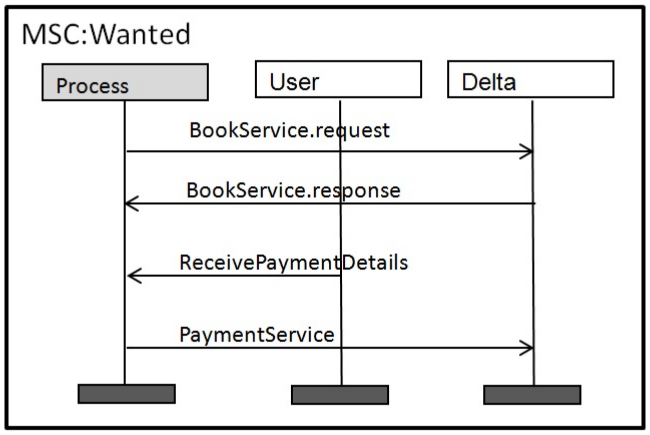


Figure 7.3: Wanted scenario (Example 3/ Property p3 of EasyTravel)

a wanted sequence. If all these messages occur, the sequence should be strictly the one depicted here. If this sequence is altered it must again be detected as a violating condition.

### 7.1.2 bMSC $\rightarrow$ Automata Transformational Algorithm

A property specified using bMSC is transformed into an automata by first extracting a subset where either the “send” event $s$ or its “receive event” $f(s)$ belongs to $E_p$. In order words, only the message interactions pertaining to the process being monitored are considered. The automata created using a bMSC is essentially a mealy machine.

**Definition 1**: A Mealy machine $M$ is a 6-tuple, $(Q, Q_0, \Sigma, \Lambda, T, G)$, consisting of the following:

- a finite set of states (Q)
- a start state (also called initial state) $Q_0$ which is an element of (Q)
- a finite set called the input alphabet ($\Sigma$)
- a finite set called the output alphabet ($\Lambda$)
- a transition function ($T : Q \times \Sigma \rightarrow Q$) mapping pairs of a state and an input symbol to the corresponding next state.
- an output function ($G : Q \times \Sigma \rightarrow \Lambda$) mapping pairs of a state and an input symbol to the corresponding output symbol.

**Definition 2**: A trace $\sigma_0, \sigma_1 ... \sigma_n$ on $\Sigma$ is accepted by M iff there is a sequence $q_0 q_1 ... q_{n+1}$ of states s.t. $q_0 \in Q_0$; and for every $0 \leq i \leq n; (q_i, \sigma_i, q_{i+1}) \in$ T, and G$(q_i, \sigma_i, q_{i+1}) \neq$ “ALARM”. The language of $M$, $L(M)$, is the set of all traces accepted by $M$.

Based on these definition the bMSC translations to an automata are described below.

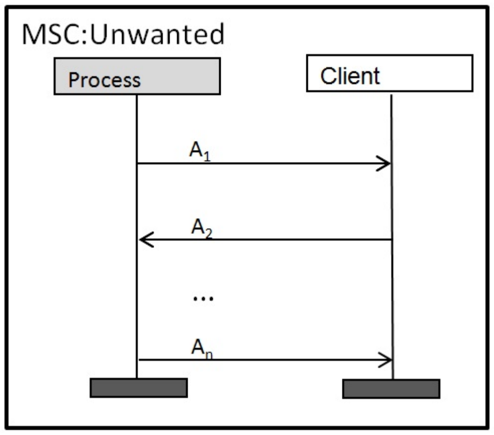


Figure 7.4: An Unwanted Scenario ($A_1- > A_2- > ...A_n$)

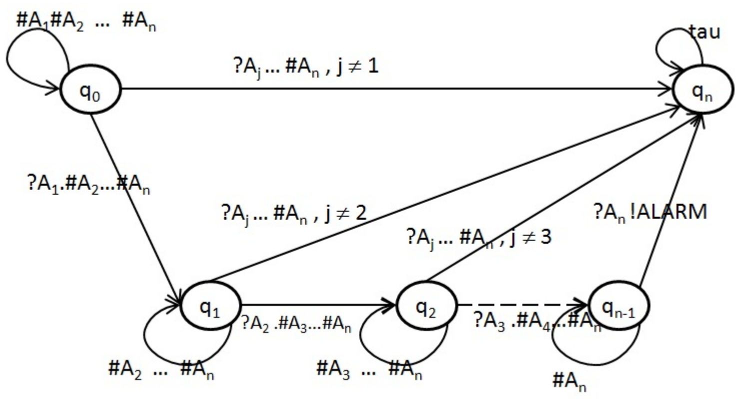


Figure 7.5: Automata for the unwanted sequence $A_1, A_2, ..., A_n$

**bMSC as an unwanted scenario:**

An unwanted scenario of a given message sequence $A_1, A_2, ..., A_n$ is shown in Figure 7.4 and the corresponding automata is shown in Figure 7.5. The number of states in this automata is $n + 1$. $q_0$ is the initial state. A state $q_i$ denotes the occurrence of message sequence $A_1, A_2, ..., A_i$. The transition relations are defined as:

$T(q_i, \#A_{i+1}.\#A_{i+2}...\#A_n) = q_i \; \forall \; i >= 0, i < n$
$T(q_i, ?A_j.\#A_{j+1}...\#A_n) = q_n \; \forall \; j \neq (i + 1), i < n$
$T(q_i, ?A_{i+1}.\#A_{i+2}...\#A_n) = q_{i+1} \; \forall \; i >= 0, i < n$

The output signal "ALARM" signifies the occurrence of violating condition. This is emitted as per the output relation defined as:

$G(q_{n-1}, ?A_n)$ = ALARM

**bMSC for an wanted scenario:**

A wanted scenario depicts a strict sequence between a set of messages. A violation against this property i.e the process terminating before the scenario is *true* results is reducing the customer rating by 10 points. Let us consider $A_1, A_2, ..., A_n$ is a wanted sequence in the bMSC of Figure 7.4 then the three messages must always follow this sequence. In case the sequence is altered or the program is terminates before all the messages have occurred, an alarm is emitted. The automata corresponding to wanted scenario of the sequence $A_1, A_2, ..., A_n$ is given in Figure 7.7. Here $q_0$ is the start state and $p_t$ is the process instance termination signal. All transitions violating the desired sequence emits an alarm. The number of states in this automata is $n + 1$. The transition relations are defined as:

$T(q_i, \#A_{i+1}.\#A_{i+2}...\#A_n) = q_i \; \forall \; i >= 0, i < n$
$T(q_i, ?A_j.\#A_{i+1}...\#A_n) = q_n \; \forall \; j \neq (i+1), i < n$
$T(q_i, ?A_{i+1}.\#A_{i+2}...\#A_n) = q_{i+1} \; \forall \; i >= 0, i < n$

The output signal is "ALARM" which signifies the violating condition. This is emitted as per the output relation defined below:

$G(q_i, ?A_j.\#A_{i+1}...\#A_n)$ = ALARM $\forall \; j \neq i + 1, i < n$

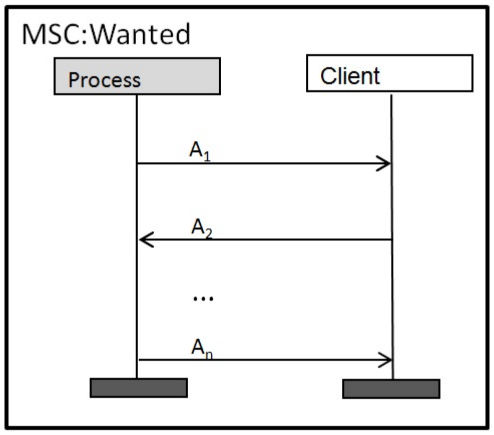


Figure 7.6: A wanted scenario ($A_1 -> A_2 -> ...A_n$)

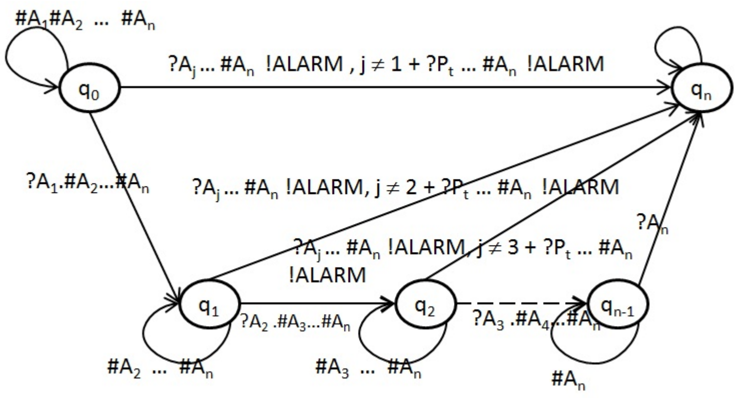

#A1#A2 ... #An
?Aj ... #An !ALARM , j ≠ 1 + ?Pt ... #An !ALARM
?A1.#A2...#An
?Aj ... #An !ALARM, j ≠ 2 + ?Pt ... #An !ALARM
?Aj ... #An !ALARM, j ≠ 3 + ?Pt ... #An !ALARM
?An
?A2 .#A3...#An
?A3 .#A4...#An
#A2 ... #An
#A3 ... #An
#An
q0
q1
q2
qn-1
qn

Figure 7.7: Automata for the wanted sequence $A_1, A_2, ..., A_n$

**bMSC as a timeout scenario:**

bMSC can also be used to specify time delays between a request/response message or any two events between the process and a client instance. Let *out* be the output message and *in* be the input message and let $x$ denote the time in milliseconds between these two messages. Figure 7.8 gives the specification of such a property using bMSC and the automata generated from it. The automata has 3 inputs *in*, *out* and *tout*. The *tout* signal corresponds to the timeout signal to be received from the environment. $q_0$ is the initial state. Two input signals may be received simultaneously. If *tout* is received before *in*, an output signal *alarm* is emitted which signifies the violating condition.

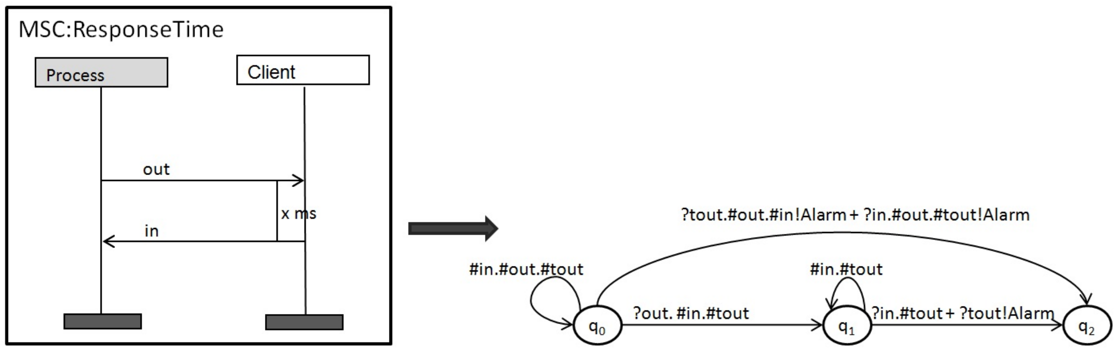


Figure 7.8: MSC and the automata corresponding to the response time property

### 7.1.3 Implementation

A mix of technologies have been used in order to develop this tool. (generating an automaton from a MSC specification) The part related to MSC specification is developed using Java based open source "Tutogef" project. It is extended to provide a user interface for creating MSCs and saving them. The automata is represented as a 2D array.

## 7.2 Using SL

### 7.2.1 Specification using SL

Monitoring Properties can also be specified using DSL (as defined in Section 3.2.1). The DSL specification for a few properties of travel process and their corresponding automata are provided below. The automata for these examples are shown in Figure 7.9 to 7.12.

With the definitions provided in Section 3.2.1, the specification of properties p6 and p7 using DSL is given below.

**p6**: The *Payment* against any service must not occur unless the service is *Booked*. Let the proposition symbol $a$ signify the *Booking.*(BookService) and the symbol $b$ signify a *Payment* (PaymentService). The property can be expressed in SL/DSL as:

$\exists x \Box ((x \equiv \bullet(a \vee x)) \wedge (b \supset x))$.

Here the auxiliary variable $x$ denotes that $a$ (*BookService*) has been true at least once in the past. Once $a$ is *true* i.e. *BookService* response is received, $x$ becomes *true* and remains true forever. Whenever $b$ is *true* implies $x$ is *true*. This property is violated if $x$ is not true when $b$ becomes true.

**p7**: Any time *PaymentService* service is called, either *Cancel Booking* has never occurred before, or *BookService* has occurred since the last occurrence of *Cancel Booking*. Let $c$ denote *PaymentService*, $b$ denote *BookService* and $a$ denote the *Cancel Booking* services. This property can then be specified as:

$\exists x, y \Box ((c \supset (x \vee y)) \wedge (x \equiv ((\neg \bullet \neg x) \wedge (\neg a))) \wedge (y \equiv (b \vee \neg \bullet \neg y) \wedge (\neg a)))))$.

Here the auxiliary variable $x$ denotes "*Cancel Booking* has never occurred" and $y$ stands for "*BookService* has occurred since the last occurrence of *Cancel Booking*".

Given below are two more examples to show how DSL can be used for specifying properties.

**Example 4**: If the service is being *booked*, then the service was *selected* but not reported to be unavailable. Let $a$ denote *Service Selection*, $b$ denote the Service Provider returning

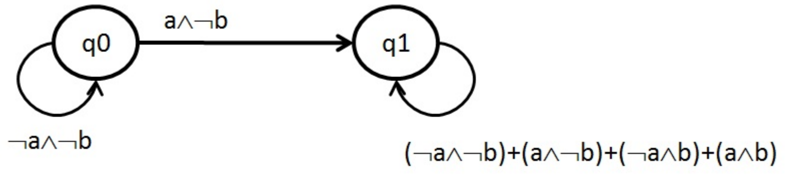


Figure 7.9: “occurrence of $a$ precedes the occurrence of $b$”(p6)

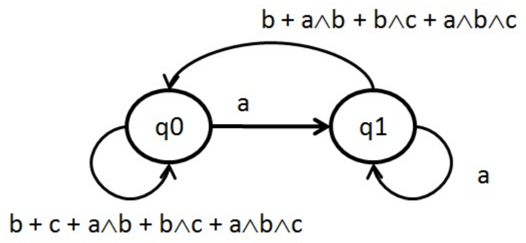


Figure 7.10: “anytime $c$ occurs either $a$ has never occurred or $b$ has occurred since the last occurrence of $a$”(p7)

the service as *unavailable*, $c$ denote *Service being booked*, and let $x$ and $y$ denote the past occurrence of $a$ and $b$. Then,
$\exists x, y \Box((c \supset (\neg y \wedge x)) \wedge (x \equiv \bullet(a \vee x)) \wedge (y \equiv \bullet((b \wedge x) \vee y)))$
is the SL specification for this property.

**Example 5**: The Service Providers *PaymentService* service must not return a payment failure against any client more than once. This will be specified as:
$\exists y \Box((a \supset \neg y) \wedge (y \equiv (\bullet(a \vee y))))$
where $a$ denotes the event of the service provider returning a failure against a payment.

## 7.2.2 Model Checking DSL

The SL formula is transformed into an automata based on the formula rewriting as explained in Section 3.2.1 and 3.2.2. Consider the property in SL given above ∃x□((x≡

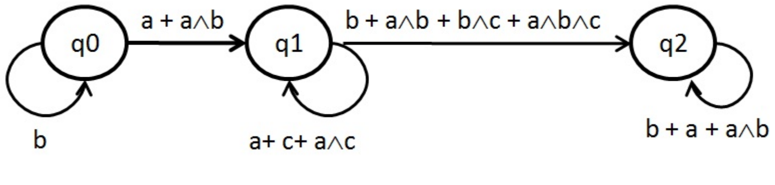


Figure 7.11: “anytime $c$ occurs it must be preceded by $a$ unless $b$ has occurred since the last occurrence of $a$”

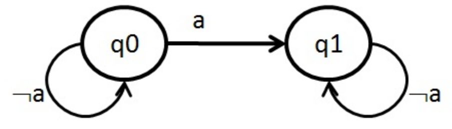


Figure 7.12: "the number of times an event $a$ occurs should be less than two"

●(a∨x))∧(b⊃x)). It means that the first time $a$ is *true* strictly preceded the first time $b$ is *true*. Here, the quantified variable $x$ stands for "$a$ has been true at least once in the past"; it is initially *false*, becomes *true* just after $a$ is *true*, and then remains *true* forever. The evaluation of the above formula for a trace ({a},{b}) is given as:

$$((x \equiv \bullet(a \vee x)) \wedge (b \supset x)) \xrightarrow{a:true} ((x \equiv \neg\bullet\neg(a \vee x)) \wedge (b \supset x)) \xrightarrow{b:true} ((x \equiv \bullet(a \vee x)) \wedge (b \supset x))$$

The given SL formula and all its derivatives represent "states". A derivative of a formula $\varphi$ can differ from $\varphi$ by the fact that some occurrences of the "●" operator are replaced by its dual "¬ ● ¬". As a consequence, a formula $\varphi$ has at most $2^{n_\varphi}$ distinct derivatives, where $n_\varphi$ is the number of "●" operators appearing in $\varphi$.

The expanded version of the automata for the above formula is given in Figure 7.13. The states representing the derivatives of the formula and the transitions are given below:

$\exists x\Box((x\equiv \bullet(a\vee x))\wedge(b\supset x)) \xrightarrow{(a\wedge b):false} \exists x\Box((x\equiv \neg\bullet\neg(a\vee x))\wedge(b\supset x))$
$\exists x\Box((x\equiv \bullet(a\vee x))\wedge(b\supset x)) \xrightarrow{(\neg a\wedge b):false} \exists x\Box((x\equiv \bullet(a\vee x))\wedge(b\supset x))$
$\exists x\Box((x\equiv \bullet(a\vee x))\wedge(b\supset x)) \xrightarrow{(a\wedge\neg b):true} \exists x\Box((x\equiv \neg\bullet\neg(a\vee x))\wedge(b\supset x))$
$\exists x\Box((x\equiv \bullet(a\vee x))\wedge(b\supset x)) \xrightarrow{(\neg a\wedge\neg b):false} \exists x\Box((x\equiv \neg\bullet\neg(a\vee x))\wedge(b\supset x))$
$\exists x\Box((x\equiv \neg\bullet\neg(a\vee x))\wedge(b\supset x)) \xrightarrow{anyinput:false} \exists x\Box((x\equiv \neg\bullet\neg(a\vee x))\wedge(b\supset x))$

The automata for property p7 is similarly depicted in Figure 7.14.

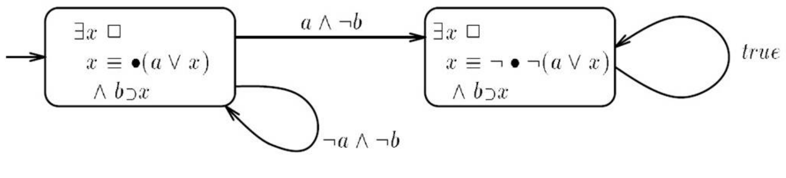


Figure 7.13: Automata for p6

It is to be noted that DSL is a syntactic fragment of SL, for the formulas of which the above process produces a deterministic automata. This is because the non determinism the automaton can only appear during the elimination of auxiliary variables. In DSL these variables appear only under a ● operator in $\varphi_i$. The value of $\varphi_i$ in each state is completely

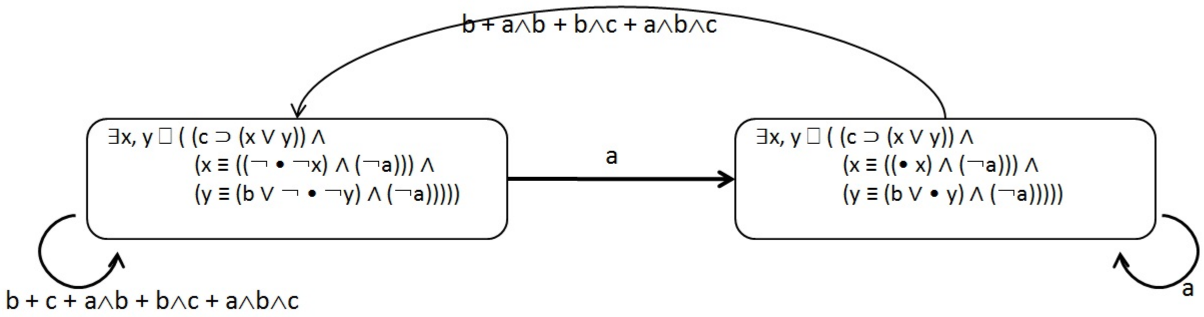

b + a∧b + b∧c + a∧b∧c
∃x, y □ ( (c ⊃ (x ∨ y)) ∧
(x ≡ ((¬ • ¬x) ∧ (¬a))) ∧
(y ≡ (b ∨ ¬ • ¬y) ∧ (¬a)))))
a
∃x, y □ ( (c ⊃ (x ∨ y)) ∧
(x ≡ ((• x) ∧ (¬a))) ∧
(y ≡ (b ∨ • y) ∧ (¬a)))))
b + c + a∧b + b∧c + a∧b∧c
a

Figure 7.14: Automata for p7

determined by the value of the propositional symbols. Since the value of $\varphi_i$ determines the value of $x_i$, the projection onto $Prop$ does not introduce non-determinism.

Let $\psi$ denote the SL formula $\exists x_1, x_2, ..., x_k \Box \varphi$, then $der(\psi)$ denotes the set of all derivatives of $\psi$. Let $PF(\psi)$ denote the past formula of $\psi$. The model checking DSL formula is provided below.

**Definition 3**: Let $\varphi$ be any general past formula. For ease of computing the truth values, the formula is broken into smaller parts denoting a $Closure(CL)$. Let $CL^{'}(\varphi)$ be the smallest set s.t.

- $\varphi \in CL^{'}(\varphi)$
- if $\neg\varphi_1 \in CL^{'}(\varphi)$ then $\varphi_1 \in CL^{'}(\varphi)$
- if $\varphi_1 \vee \varphi_2 \in CL^{'}(\varphi)$ then $\varphi_1 \in CL^{'}(\varphi)$ and $\varphi_2 \in CL^{'}(\varphi)$
- if $\varphi_1 \wedge \varphi_2 \in CL^{'}(\varphi)$ then $\varphi_1 \in CL^{'}(\varphi)$ and $\varphi_2 \in CL^{'}(\varphi)$
- if $\varphi_1 \supset \varphi_2 \in CL^{'}(\varphi)$ then $\varphi_1 \in CL^{'}(\varphi)$ and $\varphi_2 \in CL^{'}(\varphi)$
- if $\varphi_1 \equiv \varphi_2 \in CL^{'}(\varphi)$ then $\varphi_1 \in CL^{'}(\varphi)$ and $\varphi_2 \in CL^{'}(\varphi)$
- if $\bullet\varphi_1 \in CL^{'}(\varphi)$ then $\varphi \in CL^{'}(\varphi)$

Then, CL($\varphi$) = { $\varphi_1$,$\neg\varphi_1$ | $\varphi_1 \in CL^{'}(\varphi)$}

**Definition 4**: An Atom is a maximal locally consistent subset of CL($\varphi$). A set $A \subseteq$ CL($\varphi^{'}$) is called an atom of $\varphi$ such that:

- $\varphi \in A$
- $\varphi_1 \in A$ iff $(\neg\varphi_1 \notin A)$

- $(\varphi_1 \vee \varphi_2) \in A$ iff $\varphi_1 \in A$ or $\varphi_2 \in A$
- $(\varphi_1 \wedge \varphi 2) \in A$ iff $\varphi_1 \in A$ and $\varphi_2 \in A$
- $(\varphi_1 \supset \varphi_2) \in A$ iff $\varphi_1 \notin A$ or $\varphi_2 \in A$
- $(\varphi_1 \equiv \varphi_2) \in A$ iff $(\varphi_1 \notin A$ and $\varphi_2 \notin A)$ or $(\varphi_1 \in A$ and $\varphi_2 \in A)$
- $\bullet\varphi_1 \notin A$

**Definition 5**: Let *At($\psi$)* denote the set of atoms corresponding to the past formula of $\psi$ i.e. PF($\psi$) and $At_\varphi$ denote the set of atoms corresponding to $\varphi$ and all its derivatives i.e., all possible states in the automata; then,
$At_\psi \triangleq \bigcup_{\psi_1 \in der(\psi)} At(\psi_1)$
For a SL formula $\psi$ we construct an Automata A($\psi$) = (Q,$\Sigma, \delta, Q_0$,F) as below:

- Q = $At_\psi$, the set of atoms
- $\Sigma$=$2^{Prop}$, the input alphabet where *Prop* is the set of propositions (Observables)
- $Q_0$ = At($\psi$), the initial state
- F=Q, as all states in this automata are accepting
- $\delta$=Q$\times\Sigma\times$Q, the transition relation such that for A, B $\in At_\psi$, $\omega \in \Sigma$, (A,$\omega$,B) $\in \delta$ iff
  - A$\cap$Prop=$\omega$
  - $\varphi \in$A iff $\neg \bullet \neg\varphi \in$ B
  - $\neg\varphi \in$A iff $\neg \bullet \varphi \in$ B

**Definition 6**: Automata Reduction: A state in the quotient automata is represented by the collection of all atoms in At($\varphi$) where $\varphi \in der(\psi)$. Each state $S_i$ of the automata is therefore created as: $S_i = \{A \mid A \in At(\varphi_i)\}$ and,
$S_i \xrightarrow{\omega} S_j$ if $\exists A \in S_i$,$B \in S_j$ s.t. $A \xrightarrow{\omega} B$.

**Example**

Consider the SL formula $\exists x\Box((x \equiv \bullet(a \vee x)) \wedge (b \supset x))$. A stepwise explanation of automata generation is given below.
**Step 1:**
Here, $\psi = \exists x\Box((x \equiv \bullet(a \vee x)) \wedge (b \supset x))$
and *PF($\psi$)* i.e. the past formula of $\psi$ is denoted as $\varphi$=$((x \equiv \bullet(a \vee x)) \wedge (b \supset x))$
$der(\psi)$ i.e. set of derivatives of $\varphi$ are:
$\varphi_1 = ((x \equiv \bullet(a \vee x)) \wedge (b \supset x))$ and $\varphi_2 = ((x \equiv \neg \bullet \neg(a \vee x)) \wedge (b \supset x))$

**Step 2:**
Here, we find the Closures of $\varphi_1$ and $\varphi_2$. We include the formula and then break it into smaller parts as per the definition of Closures.
$CL^{'}(\varphi_1) = \{((x \equiv \bullet(a \vee x) \wedge (b \supset x)), (x \equiv \bullet(a \vee x)), (b \supset x), x, \bullet(a \vee x), (a \vee x), a, b\}$
$CL(\varphi_1) = \{((x \equiv \bullet(a \vee x)) \wedge (b \supset x)), \neg((x \equiv \bullet(a \vee x)) \wedge (b \supset x)), (x \equiv \bullet(a \vee x)), \neg(x \equiv \bullet(a \vee x)), (b \supset x), \neg(b \supset x), x, \neg x, \bullet(a \vee x), \neg\bullet(a \vee x), (a \vee x), \neg(a \vee x), a, \neg a, b, \neg b\}$
$CL^{'}(\varphi_2) = \{((x \equiv \neg\bullet\neg(a \vee x)) \wedge (b \supset x), (x \equiv \neg\bullet\neg(a \vee x), (b \supset x)), x, \neg\bullet\neg(a \vee x), (a \vee x), a, b\}$
$CL(\varphi_2) = \{((x \equiv \neg\bullet\neg(a \vee x)) \wedge (b \supset x), \neg((x \equiv \neg\bullet\neg(a \vee x)) \wedge (b \supset x)), (x \equiv \neg\bullet\neg(a \vee x)), \neg(x \equiv \neg\bullet\neg(a \vee x)), (b \supset x), \neg(b \supset x), x, \neg x, \neg\bullet\neg(a \vee x), \neg\neg\bullet\neg\neg(a \vee x), (a \vee x), \neg(a \vee x), a, \neg a, b, \neg b\}$ or,
$\{(x \equiv \neg\bullet\neg(a \vee x)) \wedge (b \supset x), \neg((x \equiv \neg\bullet\neg(a \vee x)) \wedge (b \supset x)), (x \equiv \neg\bullet\neg(a \vee x)), \neg(x \equiv \neg\bullet\neg(a \vee x)), (b \supset x), \neg(b \supset x), x, \neg x, \neg\bullet\neg(a \vee x), \bullet(a \vee x), (a \vee x), \neg(a \vee x), a, \neg a, b, \neg b\}$

**Step 3:**
Let $A_i$ denote the set of atoms for $\varphi_1$ and $B_i$ denote the set of atoms for $\varphi_2$
Then, $A_1 = \{((\mathrm{x}\equiv \bullet(\mathrm{a}\vee\mathrm{x}))\wedge(\mathrm{b}\supset\mathrm{x})), (\mathrm{x}\equiv \bullet(\mathrm{a}\vee\mathrm{x})), (\mathrm{b}\supset\mathrm{x}), \neg\bullet(a\vee\mathrm{x}), \neg\mathrm{x}, a\vee\mathrm{x}, \mathrm{a}, \neg\mathrm{b}\}$.
Here $\neg\bullet(a \vee x)$ is included as for any general formula $\varphi$, $\bullet\varphi$ cannot be in a atom. Since $\bullet(a \vee x)$ cannot be included $\neg x$ is included in order to satisfy the equivalence relation of x i.e. x≡ •(a∨x). Now, since $\neg(a \vee x) \notin A$, $(a \vee x)$ is included in A. And since $(a \vee x)$ is included and x is false, a is included. For $(b \supset x)$ since $\neg x$ is included $\neg b)$ is included.

$A_2 = \{((\mathrm{x}\equiv \bullet(\mathrm{a}\vee\mathrm{x})) \wedge (\mathrm{b}\supset\mathrm{x})), (\mathrm{x}\equiv \bullet(\mathrm{a}\vee\mathrm{x})), (\mathrm{b}\supset\mathrm{x}), \neg\bullet(a\vee\mathrm{x}), \neg\mathrm{x}, \neg(\mathrm{a}\vee\mathrm{x}), \neg\mathrm{a}, \neg\mathrm{b}\}$.
Similarly,
$B_1 = \{((\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})) \wedge (\mathrm{b}\supset\mathrm{x})), (\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})), (\mathrm{b}\supset\mathrm{x}), \neg\bullet\neg(a\vee\mathrm{x}), \mathrm{x}, a\vee\mathrm{x}, \mathrm{a}, \mathrm{b}\}$.
$B_2 = \{((\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})) \wedge (\mathrm{b}\supset\mathrm{x})), (\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})), (\mathrm{b}\supset\mathrm{x}), \neg\bullet\neg(a\vee\mathrm{x}), \mathrm{x}, a\vee\mathrm{x}, \neg\mathrm{a}, \mathrm{b}\}$.
$B_3 = \{((\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})) \wedge (\mathrm{b}\supset\mathrm{x})), (\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})), (\mathrm{b}\supset\mathrm{x}), \neg\bullet\neg(a\vee\mathrm{x}), \mathrm{x}, a\vee\mathrm{x}, \mathrm{a}, \neg\mathrm{b}\}$.
$B_4 = \{((\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})) \wedge (\mathrm{b}\supset\mathrm{x})), (\mathrm{x}\equiv \neg\bullet\neg(\mathrm{a}\vee\mathrm{x})), (\mathrm{b}\supset\mathrm{x}), \neg\bullet\neg(a\vee\mathrm{x}), \mathrm{x}, a\vee\mathrm{x}, \neg\mathrm{a}, \neg\mathrm{b}\}$

**Step 4:**
The transitions in the automata are identified by taking a projection with *Prop* with the atoms set. The resulting transitions are:
$A_1 \xrightarrow{a\wedge\neg b} B_1, B_2, B_3, B_4,$
$A_2 \xrightarrow{\neg a\wedge\neg b} A_2,$
$B_1 \xrightarrow{any\ input} B_1, B_2, B_3, B_4,$
$B_2 \xrightarrow{any\ input} B_1, B_2, B_3, B_4,$
$B_3 \xrightarrow{any\ input} B_1, B_2, B_3, B_4,$
$B_1 \xrightarrow{any\ input} B_1, B_2, B_3, B_4,$

**Step 5:**
We now need to create the quotient automata by applying the automata reduction rules. $A_1$ and $A_1$ denotes state $S_0$ and $B_1$, $B_2$, $B_3$, $B_4$ denotes state $S_1$. The transitions are:
$S_0 \xrightarrow{a\wedge\neg b} S_1$

Table 7.1: DSL Syntax in BNF

| |
|---|
| Aux:=$x_i$ |
| Prop:=a,...,z |
| PF:=Prop\|Aux\|($\neg$ PF)\|(PF $\vee$ PF)\|(PF $\wedge$ PF)\|($\bullet$ PF)\| ($\neg \bullet \neg$ PF) \| (PF $\equiv$ PF) \| (PF $\supset$ PF) |
| PPF:=Prop\|($\neg$ PPF)\|($\vee$ PPF)\|(PPF $\wedge$ PPF)\|($\bullet$ PPF)\| ($\neg \bullet \neg$ PPF)\|(PPF $\equiv$ PPF)\|(PPF $\supset$ PPF) |
| APF:=($\neg$ APF)\|(APF $\vee$ APF)\|(APF $\wedge$ APF)\|($\bullet$ PF)\| ($\neg \bullet \neg$ PF)\|(APF $\equiv$ APF)\|(APF $\supset$ APF)\| PPF |
| EqTerm:=(Aux $\equiv$ APF) |
| EqList:=EqTerm\|EqTerm $\wedge$EqList |
| SpecialPF:=(PF $\wedge$ EqList) |
| Auxlist:=Aux \| Aux,AuxList |
| DSL:=$\exists$ AuxList $\Box$ SpecialPF |

$S_0 \xrightarrow{\neg a \wedge \neg b} S_0$
$S_1 \xrightarrow{any\ input} S_1$

### 7.2.3 DSL → Automata: Transformational Algorithm

DSL to Automata transformation algorithm takes as an input any DSL formula and evaluates the formula to find the next state for all possible transitions. The final automata is represented in the form of a state transition table represented by a 2D array automata[i][t], the value of which represents the next state for a transition t occurring in state i. The automata is generated in two broad steps as described below:

#### Preprocessing

Pre-processing step involves parsing the formula and identifying syntax errors (if any). The DSL syntax in BNF notation is given in Table 7.1.

A DSL formula enables the auxiliary variables to be defined with separate past formulas. In addition, the auxiliary variable in an *APF* may only appear under a $\bullet$ operator. The syntax of DSL in BNF after eliminating left recursion and left factoring is as given in Table 7.2. As can be seen, our implementation uses a “Predictive Parser” wherein each nonterminal is defined as a procedure and it decides which production to use by looking at the lookahead token. Predictive parsers are simple, as they require no backtracking. For verifying in DSL, the condition that an auxiliary variable must occur in the past formula with a “$\bullet$” operator, we have made use of simple flags which are set and reset in the procedure for the nonterminal *Eqterm*. Apart from the syntax analysis, the pre processing step also stores useful information like set of propositions, auxiliary variables, the respective past formulas of each auxiliary variables etc.

Table 7.2: DSL Syntax in BNF after eliminating left recursion and left factoring

| |
|---|
| PF2:=$\vee$ PF)\|$\wedge$ PF)\| $\equiv$ PF)\| $\supset$ PF) |
| PF1:=$\neg$ PF)\|$\bullet$ PF)\|$\neg\bullet\neg$ PF)\|PF PF2) |
| PF:=Prop\|Aux\|(PF1 |
| PPF2:=$\vee$ PPF)\|$\wedge$ PPF)\| $\equiv$ PPF)\| $\supset$ PPF) |
| PPF1:=$\neg$ PPF)\|$\bullet$ PPF)\|$\neg\bullet\neg$ PPF)\|PPF PPF |
| PPF:=Prop\|(PPF1 |
| APF2:=$\vee$ APF)\|$\wedge$ APF)\| $\equiv$ APF)\| $\supset$ APF) |
| APF1:=$\neg$ APF)\|$\bullet$ PF)\|$\neg\bullet\neg$ PF)\|APF APF2) |
| APF:=(APF1\|PPF |
| Eqterm:=(Aux$\equiv$ APF) |
| Eqlist:=Eqterm Eqlist1 |
| Eqlist1:=$\wedge$ Eqlist\|$\epsilon$ |
| Auxlist:=Aux Auxlist1 |
| Auxlist1:=,Auxlist\| $\epsilon$ |
| SpecialPF:=(PF$\wedge$ Eqlist) |
| DSLformula:=$\exists$ Auxlist$\square$ SpecialPF |

**Automata Generation**

The states in the final automata are the set of all derivatives of the past formula. A derivative of a past formula is formed by replacing a $\bullet$ to $\neg\bullet\neg$ and vice-versa. It therefore depends on the number of past operators in the formula. The algorithm for the automata generation is given in Algorithm 1 and that for evaluation of a given state is given in Algorithm 2. *CreateDFA* evaluates the SL formula for each state and transition. In order to find the value of auxiliary variables it evaluates their respective past formulas by taking the truth value as "false" initially. The value returned is the actual truth value to be used for the evaluating that state and transition. This is because the state evaluation in the language SL depends only on the $\bullet$ and $\neg\bullet\neg$ operator. (see Section 3.2.2).

The parameter *expr* in the function *Evaluate* is the formula to be evaluated. The propositional as well as the auxiliary variables are replaced with the actual truth values as computed above. The function returns the next state depending on whether the formula evaluates to true or false. The function makes use of two stacks: one for Operands(Propositional and Auxiliary Variables) and other for Operators($\vee, \wedge, \equiv, \supset, \neg, \bullet, \neg\bullet\neg$). Operators are implemented in the form of static method calls in Java by giving their equivalent textual representation.

The generated automata is then minimized using standard DFA minimization algorithm.

**Algorithm 1** Creating DFA

```
CreateDFA($\psi$)
curr-state = $\psi$.PastFormula() i.e. $\varphi$
all-states = Derivatives($\varphi$)
all-transitions = Combinations($Prop$)
for each state in all-states do
  for each transition in all-transitions do
    $\varphi_1$ = Replace$\varphi$WithPropValuesOfTransition()
    for each aux in Aux do
      $\varphi$ = PastFormula(aux)
      $\varphi^{'}$ = Replace$\varphi$withAuxValueFalse()
      aux.value = Evaluate($\varphi^{'}$)
    end for
    $\varphi_2$ = Replace$\varphi_1$WithAuxValues()
    (ret,next-state) = Evaluate($\varphi_2$)
    if ret=true then
      automata[state][transition] = next-state
    end if
  end for
end for
MinimizeAutomata()
```

**Complexity**

If $n$ is the length of the SL formula, the preprocessing step takes time $O(n)$. Let $m$ be the number of past operators ($\bullet$ or $\neg\bullet\neg$) and $k$ be the number of propositions. The maximum number of states can then be $2^m$ and there are $2^k$ possible transitions. The evaluation of a single state and transition takes $O(n)$ time. The complexity of complete automata generation is therefore $O(2^m.2^k.n) = O(n.2^{m+k})$. If $l$ is the number of actual states computed for the formula the minimization algorithm in our current implementation takes $O(l^2.l) = O(l^3)$ time. This can be replaced by better algorithms for DFA minimization to attain *O(l log l)* time.

## 7.3 Integrating observers with workflow engine

### 7.3.1 Generating "events" for the observers

The interaction between process and the web services is nothing but the messages/events observed under the SOAP protocol. As our monitoring observer needs to work on the same signals/messages as the main program engine, we need to capture these messages. This is depicted in Figure 7.15. Let P denote the process to be monitored for properties Pr = $Pr_0$,$Pr_l Pr_m$, and let Serv = $Serv_o$, , $Serv_k$ be the set of web-services with which $P$ interacts. For $Serv_i$, let $Req_i$ = $(Req_{i_o}, t_{i_o})$ , $(Req_{i_1}, t_{i_1})$,, $(Req_{i_n}, t_{i_n})$ be the set of all

**Algorithm 2** Evaluating a state(past formula)

```
Evaluate(expr)
while not(end of expr) do
  Read next character (ch) from expr
  if $ch \in \{Aux, Prop\}$ then
    OperandStack.Push(ch)
    if $ch \in \{Oper\}$ then
      if $ch ==')'$ then
        Oper = OperatorStack.Pop()
        if $Oper \in \{\vee, \wedge, \equiv, \supset\}$ then
          Opr1 = Operandstack.Pop()
          Opr2 = Operandstack.Pop()
          Opr = Oper(Opr1,Opr2)
          OperandStack.Push(Opr)
        end if
        if $Oper \in \{\neg, \bullet, \neg\bullet\neg\}$ then
          Opr1 = Operandstack.Pop()
          Opr = Oper(Opr1)
          Opr = Oper(Opr1,Opr2)
          OperandStack.Push(Opr)
          if $(Opr1 = true) \wedge (Oper = \bullet)$ then
            $next - state = Replace \bullet with \neg \bullet \neg in Expr$
          end if
          if $(Opr1 = false) \wedge (Oper = \neg \bullet \neg)$ then
            $next - state = Replace \neg \bullet \neg with \bullet in Expr$
          end if
        end if
      end if
      if $ch = '('$ then
        OperatorStack.Push(ch)
      end if
    end if
  end if
end while
ret = $Operandstack.Pop()$
if $value = false$ then
  $next - state$ = ALARM
end if
return $< ret, next - state >$
```

messages going from P to $Serv_i$, together with the associated timeouts for receiving the response; and $Resp_i = Resp_{i_o}$,$Resp_{i_1}$,..., $Resp_{i_n}$ be the set of all messages received by P from $Serv_i$. Let $Obs_i$ denote the observer for $Pr_i$, then the events required by $Obs_i$ are given by

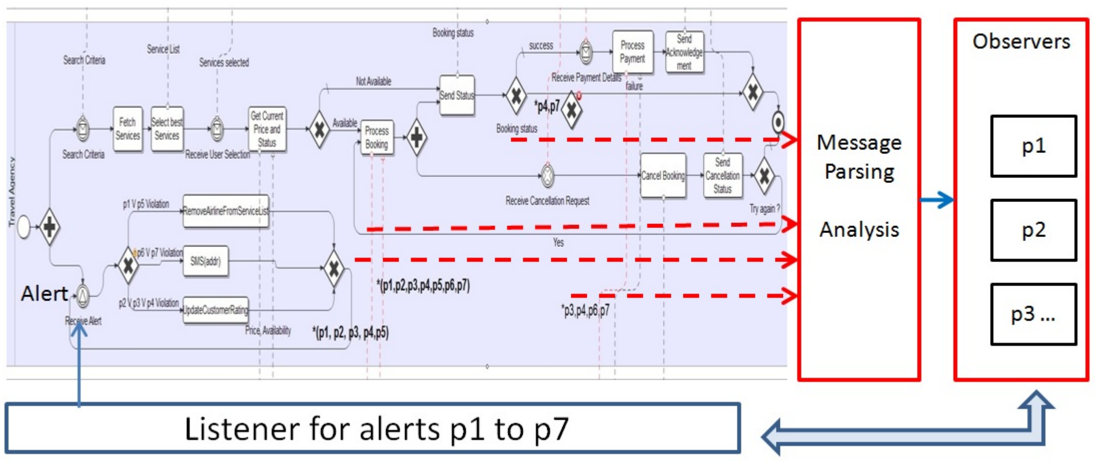


Figure 7.15: Integrating observers with workflow engine

$\bigcup_{i \in Serv_i}(Req_i \cup Resp_i)$.

A SOAP Message contains in itself all information relevant to the process state. After intercepting, the SOAP messages are parsed and analyzed to ascertain whether they need to be translated as input signals to the monitor. SOAP messages are intercepted by adding a handler (MyHandler2.class) to the default SOAP handler chain of the travel process. A process identifier *InstanceID* is also passed to the Handler using the *setHandlerConfig* method to identify various travel processes executing concurrently. *MyHandler* is derived from a Generic Handler class and routes the SOAP messages through its *handleResponse* and *handleRequest* methods. It receives the *SOAPMessageContext* object as a parameter which is parsed, analyzed and then passed to the respective observer.

```
HandlerInfo hi = new HandlerInfo();
hi.setHandlerConfig(map);
hi.setHandlerClass(MyHandler2.class);
handlerChain.add(hi);
```

In case the web service invocation is through the REST/XML then the wrapper implementation needs to generate similar events for the observer. Since the architecture is loosely coupled adding such functionality should be trivial for the specific REST service implementer.

The SOAP messages sent and received also need to be correlated. SOAP in conjunction with WS-Addressing, contains an identifier *(messageID)* and refers to a previous message through the *relatesTo* header. We use this information for correlating messages belonging to the same service instance. It is to be noted that there is a separate executing instance of the automata for each process instance. This is managed internally in our system using a *hashmap*.

### 7.3.2 Receiving “alerts” from monitors

The observer needs a mechanism to communicate to the EasyTravel for violations detected to enable it to take appropriate actions. For our implementation we have used Orc-engine (ver 1.1) as a workflow engine which is Java based but does not have any in-built interrupt mechanism of its own. In order to receive interrupts from the observer and to be able to process them on priority, we have modified the engine to create an interrupt buffer where the violations are added as soon as they are detected. Interrupt handling mechanism is implemented by adding three classes (a) an Interrupt Event class (b) a Listener class and (c) an Interrupt buffer. These are standard implementation of callback mechanism. The Interrupt event class is used by the observer to fire an event (interrupt) as soon as violation is detected On receiving any event it notifies all its listeners which then writes to the interrupt buffer. The execution mechanism is given below:

```
Execute()
[    loop ExecuteNextActivity()
] ||
[ Interrupt listener receives current value
  of p1:p7
  if !(p[i].violation) then
      execute(action(p[i]))
]
```

### 7.3.3 Actions against “alerts”

Actions against detected violations could range from sending mails/sms, database updation, invocation of an alternate web-service or terminating current process. These actions are made available in a standard library of functions. In addition to the workflow engine, “alerts” are also sent to the admin dashboard for the administrator to have a global view of the system. Following actions are provided in our system:

- *send(‘email/sms’,prop,addr):* The property violation is emailed/smsed at address provided.
- *execute(classname,methodname):* Specific actions can be implemented as Java classes. On detection of a specific violating condition the corresponding Java method is invoked.
- *halt*: Stop executing the given process instance.

# Chapter 8

# Wf_Sla_Mon: GUI Capabilities and Experimental Evaluation

This chapter provides an overview of the implementation technologies used, GUI features, some screenshots of Wf_Sla_Mon tool and the results of the performance evaluation tests that have been performed.

## 8.1 Implementation Architecture

The technologies used for the implementation of various components in this framework is provided in Figure 8.1. Most of the transformational algorithms and interfacing is achieved using Java API. The property specification in MSC is developed using the Eclipse Graphical Editing Framework (GEF). The final observers are also in the form of Java classes. Monitors are compiled into a single jar file which can be plugged into the workflow engine. The web client for this tool monitoring is deployed on Apache Tomcat web server. Java Applets are used for displaying the monitoring results online by creating a socket connection to the workflow execution engine. In case of any violations the workflow engines sends an alert on this socket. This is refreshed on the client machine immediately. Java event handlers are used for communication between the monitors and the workflow engine. The monitors communicate with the workflow engine using interrupts (event listeners) which in turns communicates with the client using the socket connection.

## 8.2 EasyTravel in Orc

The EasyTravel process is implemented using Orc as given below. The site *ReceiveSearchCriteria* (Line 11) receives a user search criteria in the form of a tuple *(price,type)*. The process then gets the list of services meeting the criteria using *FetchServices*. This is published in a list *services* (Line 12). The user is then presented with this list out of which he selects the service he requires (*user.GetUserSelection()*) in Line 4. The *Webservice* site in Line 4 returns the site corresponding to the selected service. The process then contacts

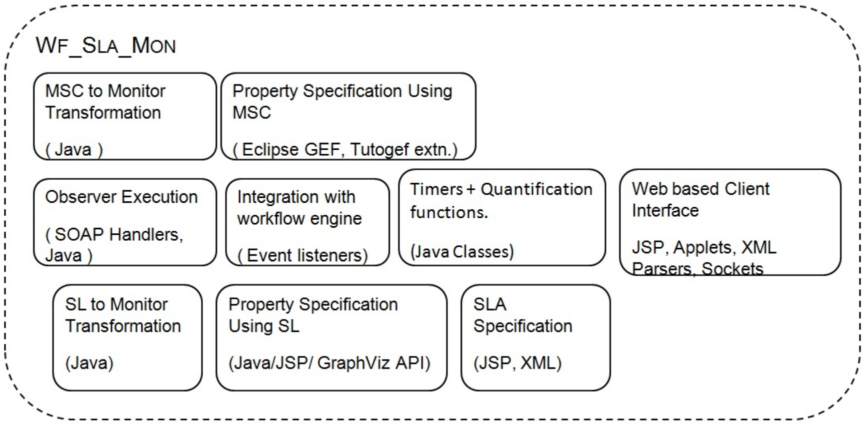

Wf_Sla_Mon
MSC to Monitor Transformation
( Java )
Property Specification Using MSC
( Eclipse GEF, Tutogef extn.)
Observer Execution
( SOAP Handlers, Java )
Integration with workflow engine
( Event listeners)
Timers + Quantification functions.
(Java Classes)
Web based Client Interface
JSP, Applets, XML Parsers, Sockets
SL to Monitor Transformation
(Java)
Property Specification Using SL
(Java/JSP/ GraphViz API)
SLA Specification
(JSP, XML)

Figure 8.1: Wf_Sla_Mon Components

this selected site (*getCurrentAvailability*) to get the current status. The service is then booked *(BookService)* and payment made *PaymentService* (Line 16). While the booking and Payment is on, in case, of a cancelation, the booking is canceled *CancelBooking*. This is indicated in the Orc process with the pruning operator < in Line 15.

```
1. def XOR(b,f,g) = if (b) then f else g
2.
3. def Select(services) =
4. user.GetUserSelection() >serviceurl> Webservice(serviceurl) >service>
5. (
6.   let(service,price) <price<
7.                             service.getCurrentAvailability()
8. )
9. >(service,price)> let(service,price)
10.
11. user.ReceiveSearchCriteria() >(price,type)>
12. travel.FetchServices(price,type)  >services>
13.   Select(services) >(first,currentprice)>
14.   (
15.     let(stat) <stat<
16.                      first.BookService() >status>
17.                           if (status=success)
18.                            then user.ReceivePaymentdDetails() >> first.PaymentService()
19.                           else sendRejectToUser()
20.                      |
21.                      first.receiveCancellation()
22.    )
23. >ret>
```

```
XOR(ret=success,sendAckToUser(),
          XOR(ret=cancel,user.CancelBooking() >> sendRejectToUser(),sendRejecttoUser())
    )
```

## 8.3 Wf_Sla_Mon: GUI Features and Capabilities

The graphical environment used in the framework has following features:

- a graphical tool for drawing MSC diagrams for specifying the properties and generating automata files from the same
- a wizard for specifying a DSL formula to be used as a property or a SLO.
- automatic generation of an executable automata from a given DSL formula
- viewing the generated automata graphically
- composing SLAs/monitors with the SLOs defined earlier
- online monitoring of all SLAs showing violations, their frequency etc.

The association of input events for the monitors(automata) is done at the time of SLA/monitor composition. One or more monitors can then be added conjunctively. A few screenshots of the tool are provided below:

**Specifying Scenarios in MSC**

The wizard for specifying properties in MSC is implemented by extending the open source "tutogef" project. The wizard generates all related monitor files. A screenshot of the tool is given in Figure 8.2

**Specifying DSL formula**

The wizard for creating DSL formula integrates the BNF grammar rules with a user interface. The user starts with the basic formula and then expands it as per the grammar rules till it reaches the terminal symbol of the grammar. This ensures that the formula is well formed. The formula can be defined without using the wizard also in which case a "Validate" button helps in verifying the syntax. Once verified, the automata can be generated and viewed graphically for easier understanding and verification. The automata is generated as a Java file which can be directly plugged-in to the Orc (or any other orchestration language like BPEL) work-flow engine. Figure 8.5 gives a compilation of some screen shots of the tool.

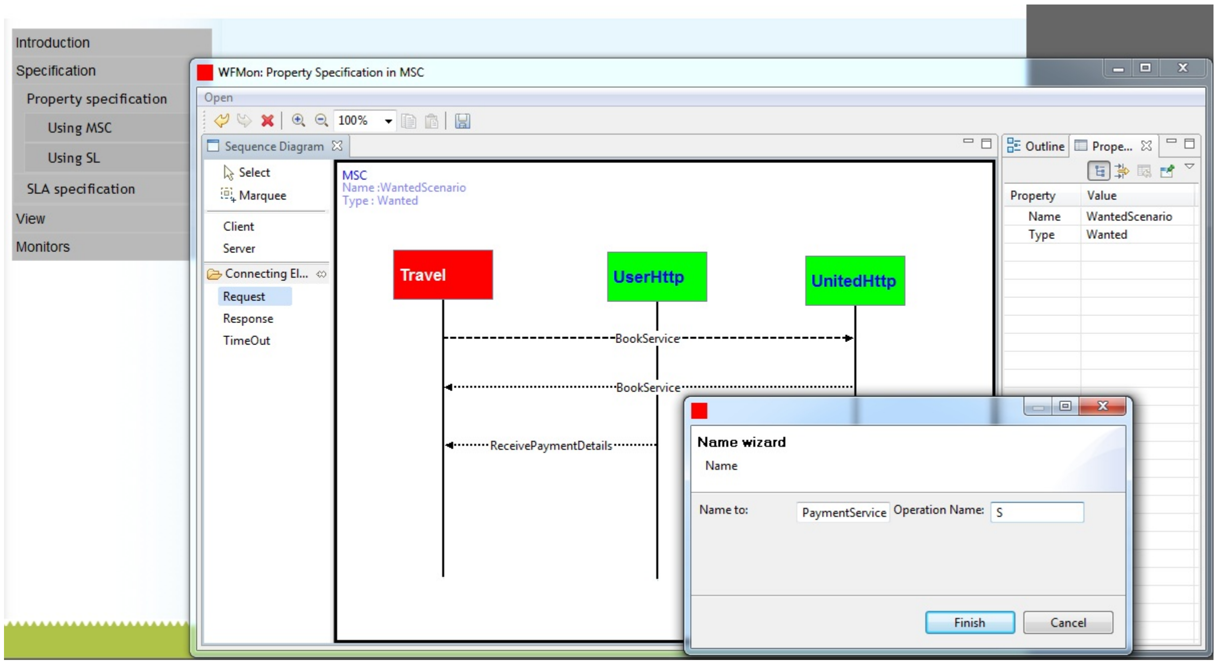
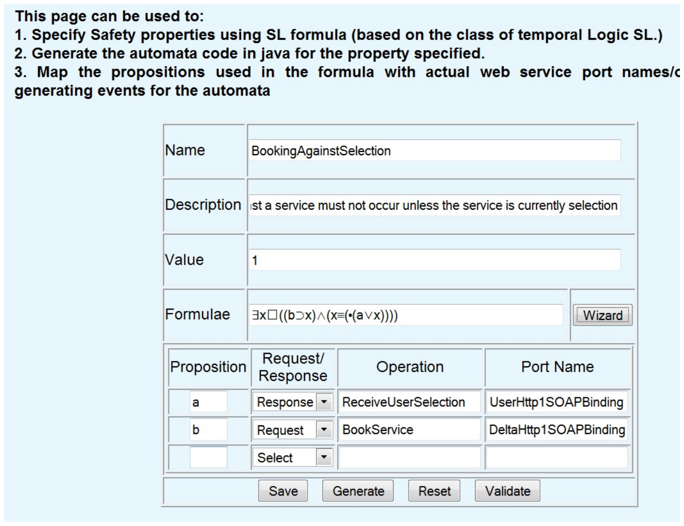


Figure 8.2: Specification of properties using MSC



**This page can be used to:**
**1. Specify Safety properties using SL formula (based on the class of temporal Logic SL.)**
**2. Generate the automata code in java for the property specified.**
**3. Map the propositions used in the formula with actual web service port names/c**
**generating events for the automata**

| | |
|---|---|
| Name | BookingAgainstSelection |
| Description | ıst a service must not occur unless the service is currently selection |
| Value | 1 |
| Formulae | ∃x□((b⊃x)∧(x≡(•(a∨x)))) |

| Proposition | Request/ Response | Operation | Port Name |
|---|---|---|---|
| a | Response | ReceiveUserSelection | UserHttp1SOAPBinding |
| b | Request | BookService | DeltaHttp1SOAPBinding |
| | Select | | |

Figure 8.3: Specification of properties using SL

### Composing SLAs from SLOs

Once the SLOs are specified as properties, SLAs are specified as a conjunction of all these SLOs along with their mapping of propositional symbols to the events in the form of ser-

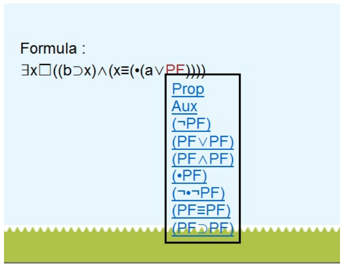

Formula :
∃x☐((b⊃x)∧(x≡(•(a∨PF))))

Figure 8.4: A wizard for specifying SL formula

vice interactions. Multiple SLAs are combined to form the global monitor specification. Let $n$ be the number of partners and let $k$ be the number of SLOs in a given SLA $SLA_i$ then,

$Monitor = SLA_1 \wedge SLA_2 \wedge ...SLA_n$
where $SLA_i = SLO_1 \wedge SLO_2...SLO_k$

The SLAs are stored in a XML format. The association of the propositional variables with the actual events corresponding to the web service interactions is done at this stage. Any property i.e. SLO, once defined, can be used in multiple SLAs by modifying its event associations.

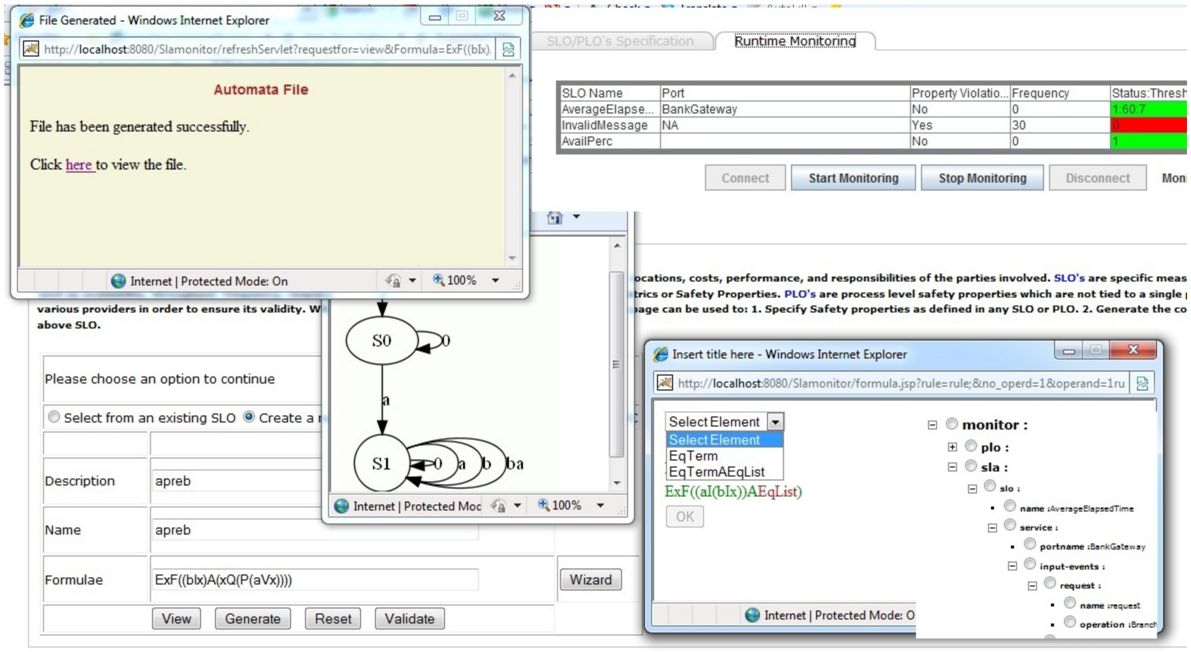
**Admin dashboard for online monitoring**

The admin dashboard is a web-based client which is developed using Java Sockets. In case of violations the monitors send events to this socket (if open). The web client is based on applets which are refreshed as soon as data is received from observers.

## 8.4 Experimental Evaluation

### 8.4.1 DSL Automata generation v/s Lustre

For evaluation of algorithm for generating the automata, monitors (automata) were created by specifying the DSL formulas and then evaluated against two parameters (i) number of states/transitions in the generated automata (ii) time taken for automata generation. The automata was compared with that of Lustre by specifying the equivalent property in Lustre and calculating the time taken in generating the automata. The results of this experiment is given in Table 8.1. The indicated time is in milliseconds and is the difference in the time returned by *System.currentTimeMillis()* command at the start and end of the automata creation program. Here, the examples refer to the examples in Section 7.2.1 except “Example 6” which was a sample case taken for increasing the number of states/transitions. The number of states and transitions created using the tool for these examples was exactly the same as that of Lustre. (Experiments with other independent


File Generated - Windows Internet Explorer

http://localhost:8080/Slamonitor/refreshServlet?requestfor=view&Formula=ExF((blx).

**Automata File**

File has been generated successfully.

Click here to view the file.

Internet | Protected Mode: On 100%

| SLO Name | Port | Property Violatio... | Frequency | Status:Thresh |
|---|---|---|---|---|
| AverageElapse... | BankGateway | No | 0 | 🟩 1:60:7 |
| InvalidMessage | NA | Yes | 30 | 🟥 0 |
| AvailPerc | | No | 0 | 🟩 1 |

ocations, costs, performance, and responsibilities of the parties involved. SLO's are specific meas

trics or Safety Properties. PLO's are process level safety properties which are not tied to a single

various providers in order to ensure its validity. W

age can be used to: 1. Specify Safety properties as defined in any SLO or PLO. 2. Generate the co

above SLO.

Please choose an option to continue

Select from an existing SLO [✓ Create a]

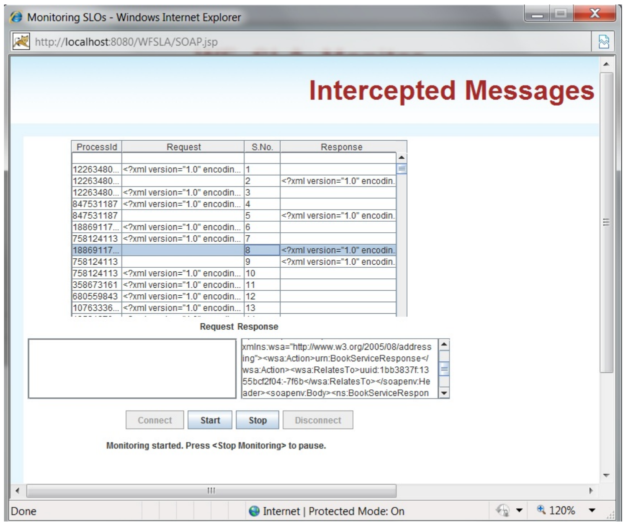
| | |
|---|---|
| Description | apreb |
| Name | apreb |
| Formulae | ExF((blx)A(xQ(P(aVx)))) |

Insert title here - Windows Internet Explorer

http://localhost:8080/Slamonitor/formula.jsp?rule=rule;&no_operd=1&operand=1ru

ExF((al(blx))AEqList)

Figure 8.5: Composing monitors/SLAs and viewing the generated automata


Monitoring SLOs - Windows Internet Explorer

http://localhost:8080/WFSLA/SOAP.jsp

**Intercepted Messages**

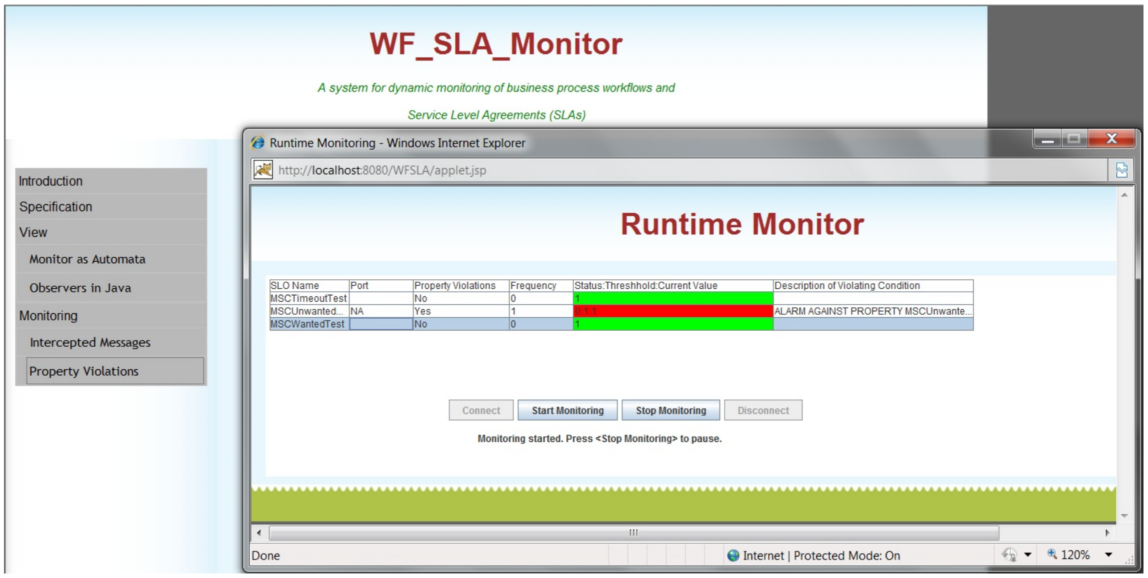
| ProcessId | Request | S.No. | Response |
|---|---|---|---|
| | | | |
| 12263480... | <?xml version="1.0" encodin... | 1 | |
| 12263480... | | 2 | <?xml version="1.0" encodin. |
| 12263480... | <?xml version="1.0" encodin... | 3 | |
| 847531187 | <?xml version="1.0" encodin... | 4 | |
| 847531187 | | 5 | <?xml version="1.0" encodin. |
| 18869117... | <?xml version="1.0" encodin... | 6 | |
| 758124113 | <?xml version="1.0" encodin... | 7 | |
| 18869117... | | 8 | <?xml version="1.0" encodin. |
| 758124113 | | 9 | <?xml version="1.0" encodin. |
| 758124113 | <?xml version="1.0" encodin... | 10 | |
| 358673161 | <?xml version="1.0" encodin... | 11 | |
| 680559843 | <?xml version="1.0" encodin... | 12 | |
| 10763336... | <?xml version="1.0" encodin... | 13 | |

Request Response

```
xmlns:wsa="http://www.w3.org/2005/08/address
ing"><wsa:Action>urn:BookServiceResponse</
wsa:Action><wsa:RelatesTo>uuid:1bb3837f:13
55bcf2f04:-7f6b</wsa:RelatesTo></soapenv:He
ader><soapenv:Body><ns:BookServiceRespon
```

Monitoring started. Press <Stop Monitoring> to pause.

Figure 8.6: Intercepted Messages


WF_SLA_Monitor
A system for dynamic monitoring of business process workflows and Service Level Agreements (SLAs)
Introduction
Specification
View
Monitor as Automata
Observers in Java
Monitoring
Intercepted Messages
Property Violations
Runtime Monitoring - Windows Internet Explorer
http://localhost:8080/WFSLA/applet.jsp
Runtime Monitor
Connect
Start Monitoring
Stop Monitoring
Disconnect
Done
Internet | Protected Mode: On
120%

| SLO Name | Port | Property Violations | Frequency | Status:Threshhold:Current Value | Description of Violating Condition |
|---|---|---|---|---|---|
| MSCTimeoutTest | | No | 0 | 1 | |
| MSCUnwanted... | NA | Yes | 1 | 0:1:1 | ALARM AGAINST PROPERTY MSCUnwante... |
| MSCWantedTest | | No | 0 | 1 | |

Monitoring started. Press <Stop Monitoring> to pause.

Figure 8.7: Admin dashboard of WF_SLA_MON

Table 8.1: Comparison of Automata with Lustre

| Formula | Time Taken (ms) | Number of States /Transitions | Number of States/Transitions (Lustre) |
|---|---|---|---|
| p6 | 2.5 | 2/3 | 2/3 |
| p7 | 8.5 | 2/4 | 2/4 |
| Example 4 | 8.5 | 3/5 | 3/5 |
| Example 5 | 2.5 | 2/3 | 2/3 |
| Example 6 | 29.5 | 4/10 | 4/10 |

examples also generated comparable number of states). Performance wise the time taken was found to increase with the number of auxiliary variables used. However, this does not have an impact as these are compiled away.

### 8.4.2 Performance impact of monitoring on executing processes

The performance impact of the monitoring engine on the execution of the process was studies. EasyTravel process was implemented as an Orc script as given earlier. Orc-engine (ver 1.1) was used as the workflow engine. External web services were implemented as java web services and deployed on Apache Tomcat on the Local Area Network to simulate this process. The time taken by the travel process for 30 executions with and without monitoring was observed. Monitoring was done by adding 1 to 6 monitors. The results

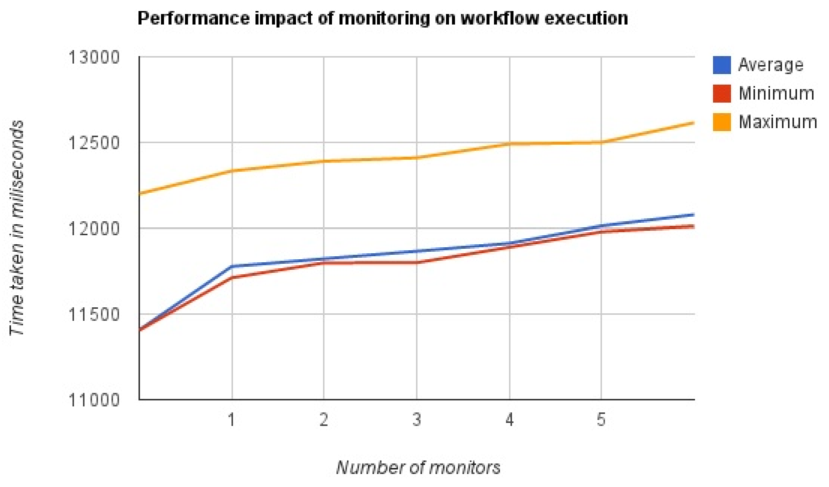


Figure 8.8: Performance impact of monitoring on workflows

are given in Figure 8.8. Recovery actions were added for sending email alerts. It was observed that monitoring increased the overall time of execution by around 200-300 ms. However this includes the time taken for recovery action. The startup time for monitor configuration was observed to be 30 ms. As the monitors can be conjunctively added, the number of monitors did not impact the performance significantly.

# Chapter 9

# Conclusions, Specific Contributions, Limitations and Future Work

## 9.1 Conclusions

Workflow technologies are most important for the automation of business processes these days. There is a growing requirement for having powerful frameworks that can allow development of an efficient, correct and inter-operable process workflow. However, there remains some gaps in conceptualizing and implementation due to the dissimilarity in the domain expertise of the people involved and the languages to be used. The business analysts try to visualize the system at a higher level and are not concerned about the implementation environment. The conceptual language they use often does not allow them to harness the power of the underlying workflow technologies/languages used. Another challenge in realizing such complex workflows is to ensure that their behavior remains consistent with the intended specifications ranging from performance requirements like QoS attributes and real-time constraints (safety properties). The workflows, thus, needs a mechanism for runtime monitoring as it is difficult to perceive all possible outcomes of its component service at the design time and evaluate dynamic executions. Further, workflows may require certain attributes that can only be computed externally. They also need to monitor their Service Level Agreements (SLAs) with partners in order to guarantee their process behavior. This thesis, tries to address a few of these problems.

BPMN2ORC, a system developed as part of this work, takes as an input a conceptualized BPMN model, validates it and then transforms it into an executional language Orc. This extends the capabilities of the language ORC to the business analysts community and the cloud computing environments also. The tool BPMN2ORC
(a) allows the business analyst community to harness the power of the language Orc.
(b) provides support for verification and debugging of BPMN models to detect concurrency properties like deadlocks etc., and
(c) can provide an intuitive interface for realizing map-reduce enabled workflows to the scientific community. It is shown that such a translation is indeed possible as it supports

translation of models with cycles, quasi-structured flows and certain unstructured flows also. Since it follows a graph based approach any translation of a BPMN gateway that depends on some parameters of another gateway can be done easily.

Further, the language Orc is studied in depth and analyzed for various workflow patterns. Its suitability is compared with other languages in this domain. The analysis shows that it is a powerful and highly expressive language supporting most of the forty three control flow patterns. However, since the language syntax is not very intuitive for the non-technical people, its usage with the business community has remain limited. The results of this study is also provided.

Another system developed as part of this work is an online monitoring system named Wf_Sla_Mon. It works by maintaining a watch on all external interactions of an executing workflow instance in order to ensure that a set of given properties are always satisfied. The system makes use of Message Sequence Charts (MSCs) and the temporal logic, SL, for specification of monitoring properties. MSCs are used to depict a wanted or unwanted scenario and SL is used to express real time constraints like safety properties. From the specification, monitors are generated automatically in the form of an automata which is wrapped with timers and other quantification attributes to form the final observer. The developed system has many add-on features for ease of monitoring. It provides a web based user interface for viewing all external message exchanges and the properties being monitored at runtime. It provides a rich GUI for specifying monitoring properties in the form of MSC based scenarios and a wizard for specifying a SL formula. The wizard not only allows specification but also validates the SL formula syntactically. The generated automata can also be viewed graphically. A user interface for specifying a complete Service Level Agreement (SLA), formally is also provided.

The generated observers are then integrated with the runtime system of the workflow engine to allow process adaptations. The process can initiate recovery actions which may be simple actions like sending emails or a more complicated ones where the process halts or takes an alternate course (use alternate web service). This allows the workflow process to be adaptive in response to the changing environment continuously.

As part of the work, it is further demonstrated, how, a given SLA can be broken into measurable and verifiable properties and then monitored by this system. The framework, thus, allows an organization to ensure that its SLAs are indeed adhered to.

Finally, experiments were conducted to study the performance impact of runtime monitoring on process executions. The study shows that adding properties do not affect the performance of the executing process significantly. (cf. Table 8.8) As the approach is conjunctive, it is scalable as demonstrated by the experiments.

## 9.2 Specific Contributions

The main contributions of this thesis are:

- A system named Bpmn2Orc, that takes as an input a conceptualized BPMN model, validates it and then transforms it into an executional language Orc (a recent language for web orchestration). Graph based transformation techniques are used wherein a given Business Process Diagram (BPD) diagram is validated for their well-formed ness and then converted to a set of Orc computation structures (in the form of Orc Graphs). These Orc Graphs are validated against certain deadlock conditions. Later, it is traversed to generate the executable code that can be used by the technical analysts for realizing the process workflow. The system Bpmn2Orc thus provides a validated code generation based on Orc starting from a BPMN specification.

- An analysis of the language Orc for realizing the forty three workflow control flow patterns and its comparison with other BPM languages.

- An online monitoring system named Wf_Sla_Mon for runtime monitoring of business process workflows. It monitors properties that are specified using temporal logic SL and generates observers in the form of a deterministic finite state automata. Properties can also be specified using MSC in the form of wanted or unwanted scenarios. The properties specified above are translated into observers, in the form of a deterministic automata, directly using the algorithm provided. The system provides a rich user interface and wizards for specification and validations of these properties. It also provides a provision to view all external message interactions and property evaluations at runtime.

- A mechanism for realizing adaptive workflows. This is achieved by integrating the runtime monitoring components (the final observers) generated earlier with the runtime system of the workflow engine. The observers generates "alerts" on detection of any violations that are then received by the workflow process. The process then initiates recovery actions as per specification. This adaptation is also illustrated as part of the work.

- Illustration of the methodology where formal properties can be composed from informal Service Level Agreements (SLAs), formally specified and monitored using the above framework.

- A study on performance impact of adding observers to the executing workflow processes. The experiments show that observers can be added conjunctively and does not have significant impact on the performance of the executing process.

## 9.3 Limitations and Future Work

It is envisaged that the tool Bpmn2Orc will allow the business community to harness the power of the language Orc through a translation mechanism. The transformation engine translates the BPMN model to Orc. At present, only the core subset of BPMN is targeted, which is for the control-flow aspects of the workflow. There are other constructs of BPMN, like, data-flow that have not been addressed here. The initial experiments show that Bpmn2Orc can provide an intuitive interface for realizing map-reduce enabled workflows

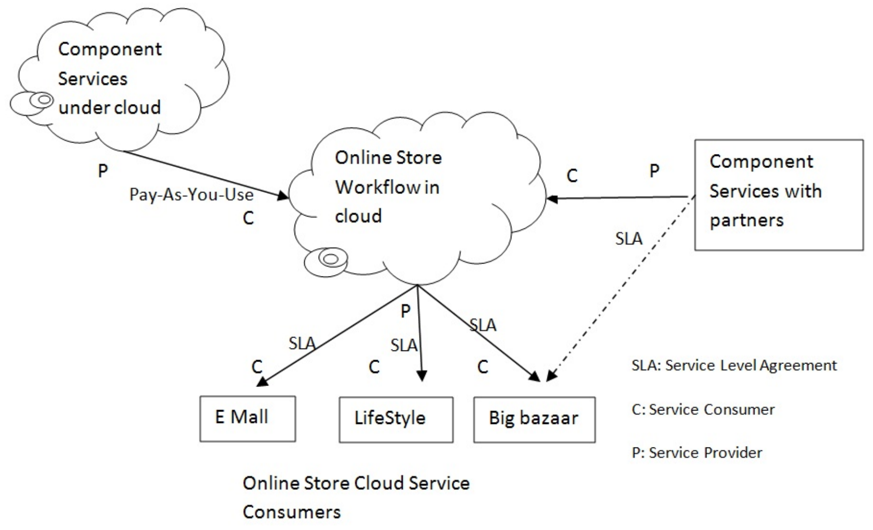

Component Services under cloud
P
Pay-As-You-Use
C
Online Store Workflow in cloud
C
P
Component Services with partners
SLA
SLA
P
SLA
SLA
C
C
C
E Mall
LifeStyle
Big bazaar
Online Store Cloud Service Consumers
SLA: Service Level Agreement
C: Service Consumer
P: Service Provider

Figure 9.1: Workflows as SaaS under Cloud

to the scientific community as scientific workflows are more data-centric and the language Orc has been applied in realizing map-enabled workflows also. Translation of dataflow aspects of the BPMN is one of the future scope of this work. Further, more experiments need to be conducted on using this framework for realizing map-reduce enabled workflows.

A translation between any source and target model typically limits the usage of target language to the constructs supported by the source language. The workflow control flow patterns based analysis of Orc shows that the language Orc has certain features that may not be directly specified using the existing BPMN syntax. The language BPMN therefore needs to be extended to utilize the full power of Orc. Extension of the language BPMN for such Orc constructs needs to be looked into as part of future work.

With the advances in cloud computing technologies, organizations are increasingly considering deploying the workflows on a cloud. Utilizing cloud services for business workflows can be achieved in following ways (a) IaaS i.e. Infrastructure As A Service e.g. using hardware, storage from the cloud environment but building, deploying and running workflow applications themselves (b) PaaS i.e Platform As A Service where the platform e.g. operating system, web application server etc is provided on a cloud and the clients can execute their workflow engines on this platform (c) SaaS i.e. Software As A Service, where, the workflow engine is provided under a cloud and the clients can create and deploy their business processes. Sometimes SaaS based workflows may be deployed for multiple tenants (same workflow specification for multiple tenants). In such a case the data needs to be segregated and security enforced. SaaS based workflows require multiple levels of SLAs to be satisfied. It is felt that the WF_SLA_MON system can be extended for such SaaS based workflows on clouds where multiple tenants access the same workflow. Monitoring support can be added by each tenant based on his requirements. It is planned to experiment this on a larger cloud based workflow which can support monitoring of SLAs.

# References

[Aalst et al., 2008] Aalst, W. M. P. v. d., Dumas, M., Ouyang, C., Rozinat, A., and Verbeek, E. (2008). Conformance checking of service behavior. *ACM Transactions on Internet Technology*, 8(3):13:1–13:30.

[Barbon et al., 2006] Barbon, F., Traverso, P., Pistore, M., and Trainotti, M. (2006). Run-time monitoring of instances and classes of web service compositions. In *International Conference on Web Services, 2006 (ICWS'06)*, pages 63 –71.

[Baresi et al., 2010] Baresi, L., Caporuscio, M., Ghezzi, C., and Guinea, S. (2010). Model-driven management of services. In *2010 IEEE 8th European Conference on Web Services (ECOWS)*, pages 147 –154.

[Baresi et al., 2009] Baresi, L., Guinea, S., Pistore, M., and Trainotti, M. (2009). Dynamo + astro: An integrated approach for bpel monitoring. In *IEEE International Conference on Web Services, 2009 (ICWS'09)*, pages 230 –237.

[Chen et al., 2011] Chen, C., Zaidman, A., and Gross, H.-G. (2011). A framework-based runtime monitoring approach for service-oriented software systems. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, QASBA '11, pages 17–20, New York, NY, USA. ACM.

[Comuzzi et al., 2009] Comuzzi, M., Kotsokalis, C., Spanoudakis, G., and Yahyapour, R. (2009). Establishing and monitoring slas in complex service based systems. In *Proceedings of the 2009 IEEE International Conference on Web Services*, ICWS '09, pages 783–790, Washington, DC, USA. IEEE Computer Society.

[Cook et al., 2006] Cook, W. R., Patwardhan, S., and Misra, J. (2006). Workflow patterns in Orc. In *Proc. of the International Conference on Coordination Models and Languages (COORDINATION)*.

[Czarnecki and Helsen, 2003] Czarnecki, K. and Helsen, S. (2003). Classification of Model Transformation Approaches.

[Fei et al., 2009] Fei, X., Lu, S., and Lin, C. (2009). A mapreduce-enabled scientific workflow composition framework. In *IEEE International Conference on Web Services, 2009 (ICWS'09)*, pages 663 –670.

[Gan et al., 2007] Gan, Y., Chechik, M., Nejati, S., Bennett, J., O'Farrell, B., and Waterhouse, J. (2007). Runtime monitoring of web service conversations. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 42–57, New York, NY, USA. ACM.

[Group, 2006] Group, O. M. (2006). Business process modeling notation (bpmn) version 1.0. omg final adopted specification.

[Haiteng et al., 2011] Haiteng, Z., Zhiqing, S., and Hong, Z. (2011). Runtime monitoring web services implemented in bpel. In *2011 International Conference on Uncertainty Reasoning and Knowledge Engineering (URKE)*, volume 1, pages 228 –231.

[Halbwachs et al., 1993] Halbwachs, N., Fernandez, J.-C., and Bouajjani, A. (1993). An executable temporal logic to express safety properties and its connection with the language lustre. In *Universite Laval*.

[Hallé and Villemaire, 2009] Hallé, S. and Villemaire, R. (2009). Runtime monitoring of web service choreographies using streaming xml. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 2118–2125, New York, NY, USA. ACM.

[Hofstede, 2005] Hofstede, A. H. M. T. (2005). Yawl: Yet another workflow language. *Information Systems*, 30:245–275.

[Keller and Ludwig, 2003] Keller, E. and Ludwig, H. (2003). The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:2003.

[Khaxar et al., 2009] Khaxar, M., Jalili, S., Khakpour, N., and Jokhio, M. (2009). Monitoring safety properties of composite web services at runtime using csp. In *Enterprise Distributed Object Computing Conference Workshops, 2009 (EDOCW '09)*, pages 107 –113.

[Kiepuszewski et al., 2000] Kiepuszewski, B., Hofstede, A. H. M. t., and Bussler, C. (2000). On structured workflow modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, CAiSE '00, pages 431–445, London, UK, UK. Springer-Verlag.

[Kitchin et al., 2009] Kitchin, D., Quark, A., Cook, W., and Misra, J. (2009). The orc programming language. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, FMOODS '09/FORTE '09, pages 1–25, Berlin, Heidelberg. Springer-Verlag.

[Löwgren, 1988] Löwgren, J. (1988). History, state and future of user interface management systems. *SIGCHI Bull.*, 20(1):32–44.

[Ludwig et al., 2004] Ludwig, H., Dan, A., and Kearney, R. (2004). Cremona: an architecture and library for creation and monitoring of ws-agreents. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, ICSOC '04, pages 65–74, New York, NY, USA. ACM.

[Mahbub and Spanoudakis, 2004] Mahbub, K. and Spanoudakis, G. (2004). A framework for requirents monitoring of service based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, ICSOC '04, pages 84–93, New York, NY, USA. ACM.

[Misra, 2006] Misra, J. (2006). Computation orchestration: A basis for wide-area computing. In *Journal of Software and Systems Modeling*, pages 10–1007.

[Morgan et al., 2005] Morgan, G., Parkin, S., Molina-jimenez, C., and Skene, J. (2005). Monitoring middleware for service level agreements in heterogeneous environments. In *In Proceedings of the 5th IFIP Conference on e-Commerce, e-Business, and e-Government (I3E*, pages 26–28.

[Moser et al., 2008] Moser, O., Rosenberg, F., and Dustdar, S. (2008). Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 815–824, New York, NY, USA. ACM.

[Ouyang et al., 2009] Ouyang, C., Dumas, M., Aalst, W. M. P. V. D., Hofstede, A. H. M. T., and Mendling, J. (2009). From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology*, 19(1):2:1–2:37.

[Ouyang et al., 2007] Ouyang, C., Dumas, M., ter Hofstede, A. H., and van der Aalst, W. M. (2007). Pattern-based translation of bpmn process models to bpel web services. *International Journal of Web Services Research (JWSR)*, 5(1):42–62.

[Papazoglou et al., 2007] Papazoglou, M. P., Traverso, P., Ricerca, I., and Tecnologica, S. (2007). Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40:2007.

[Raimondi et al., 2008] Raimondi, F., Skene, J., and Emmerich, W. (2008). Efficient online monitoring of web-service slas. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 170–180, New York, NY, USA. ACM.

[Russell et al., 2006] Russell, N., Hofstede, A. H. M. T., and Mulyar, N. (2006). Workflow controlflow patterns: A revised view. Technical report, BPM Center Report BPM-06-22.

[Sahai et al., 2002] Sahai, A., Machiraju, V., Sayal, M., Moorsel, A. P. A. v., and Casati, F. (2002). Automated sla monitoring for web services. In *Proceedings of the 13th*

*IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications*, DSOM '02, pages 28–41, London, UK, UK. Springer-Verlag.

[Schmid, 2011] Schmid, M. (2011). An approach for autonomic performance management in soa workflows. In *2011 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 698 –701.

[Stohr and Zhao, 2001] Stohr, E. A. and Zhao, J. L. (2001). Workflow automation: Overview and research issues. *Information Systems Frontiers*, 3:281–296. 10.1023/A:1011457324641.

[The University of Texas, 2011] The University of Texas (2011). Orc user guide. `http://orc.csres.utexas.edu/documentation/html/userguide/userguide.html`.

[van der Aalst and vanHee, 2004] van der Aalst, W. and vanHee, K. (2004). *Workflow Management*. MIT Press.

[van der Aalst, 1998] van der Aalst, W. M. P. (1998). The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66.

[Van Der Aalst et al., 2003] Van Der Aalst, W. M. P., Ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51.

[White, 2005] White, S. (2005). Using bpmn to model a bpel process. Technical report, BPTrends, 3(3):118.

[Wohed et al., 2009] Wohed, P., Russell, N., ter Hofstede, A. H. M., Andersson, B., and van der Aalst, W. M. P. (2009). Patterns-based evaluation of open source bpm systems: The cases of jbpm, openwfe, and enhydra shark. *Information and Software Technology*, 51(8):1187–1216.

[Wohed et al., 2006] Wohed, P., van der Aalst, W. M. P., Dumas, M., Arthur, and Russell, N. (2006). Pattern-based Analysis of BPMN - An extensive evaluation of the Control-flow, the Data and the Resource Perspectives. Technical Report BPM-06-17, BPM Center.

[Wohed et al., 2002] Wohed, P., van der Aalst, W. M. P., Dumas, M., and ter Hofstede, A. H. (2002). Pattern based analysis of bpel4ws. Technical report, Technical report, FIT-TR-2002-04, QUT.

[Wohed et al., 2003] Wohed, P., van der Aalst, W. M. P., Dumas, M., and ter Hofstede, A. H. (2003). Analysis of web services composition languages: The case of bpel4ws. In *PROC. OF ER03, LNCS 2813*, pages 200–215. Springer Verlag.

[Wu et al., 2011] Wu, G., Wei, J., Ye, C., Shao, X., Zhong, H., and Huang, T. (2011). Runtime monitoring of data-centric temporal properties for web services. In *2011 IEEE International Conference on Web Services (ICWS)*, pages 161 –170.

[Yang et al., 2010] Yang, Q., Ma, D., Zhao, Y., and Li, Z. (2010). Towards a formal verification approach for implementation of web services specifications. In *Proceedings of the 2010 IEEE Asia-Pacific Services Computing Conference*, APSCC '10, pages 269–276, Washington, DC, USA. IEEE Computer Society.

[Zhang and Li, 2010] Zhang, G. and Li, B. (2010). A way to model flow construct and its three properties verification for bpel specification. In *Proceedings of the 2010 IEEE Asia-Pacific Services Computing Conference*, APSCC '10, pages 277–284, Washington, DC, USA. IEEE Computer Society.

# List of Publications

[Peer Reviewed International Conference Publications]

1. N Goel and R K Shyamasundar. *Automatic Monitoring of SLAs of Web Services.* In IEEE Asia Pacific Services Computing Conference, APSCC 2010: 99-106.

2. Nihita Goel, N. V. Narendra Kumar, R. K. Shyamasundar. *SLAMonitor: A System for Dynamic Monitoring of Adaptive Web Services..* In IEEE European Conference on Web Services, ECOWS 2011: 109-116.

3. N Goel and R K Shyamasundar. *An executional framework of BPMN using Orc.* In IEEE Asia Pacific Services Computing Conference, APSCC 2011.

[Journal Publications]

1. N Goel and R K Shyamasundar. *An executional framework of BPMN using Orc.* In FTRA Journal of Convergence, Vol 3, No 1, Mar 15, 2012.

2. [**Submitted**] N Goel and R K Shyamasundar. WF_SLA_MON*: A system for runtime monitoring of business process workflows.* In IEEE Transaction Services Computing.

# Biography of Candidate

Nihita Goel has a Masters in Computer Science and around 17 years of experience in the field of Information Technology. She is presently working as Scientific Officer and Head, Information Systems Development Group at the Tata Institute of Fundamental Research, India and also pursuing her PhD under the guidance of Prof R.K Shyamasundar. She can be reached at nihita@tifr.res.in

## Biography of Supervisor

R.K. Shyamasundar is a Fellow IEEE and Fellow ACM holds M.E., and Ph.D in Computer Science and Automation from IISc, Bangalore. He is currently Senior Professor and JC Bose National Fellow at the Tata Institute of Fundamental Research, where he served also as the founder Dean of the School of Technology and Computer Science. His principal areas of interest are concurrent programming languages, formal methods, realtime systems and information security. He has more than 250 publications in peer reviewed journals and conferences, and several international patents in US and India. Thirty five students have done their Ph.D. under his guidance in India and US and has served on IEEE Standards Committee. He did his post-doctoral work during 1978-1979 as an International Research Fellow at Eindhoven Technological University, Eindhoven, Netherlands under the famed Professor Dr. Edsgar W Dijkstra. He has been on the Faculty/Staff at IBM TJ Watson Research center, Eindhoven University of Technology, State University of Utrecht, Pennsylvania State University, University of Illinois at Urbana, University of California, San Diego, ENSMP Sophia Antipolis, IRISA, Rennes, Verimag Grenoble Max Planck Institute for Computer Science at Saarbrucken, Ericsson Fellow at Univ. of Linkoping etc. He is a Fellow of all the Indian National Academies of Science and Engineering and also a Fellow of the Academy of Sciences of the Developing world (TWAS) at Trieste, Italy. He can be reached at shyam@tcs.tifr.res.in (http://www.tcs.tifr.res.in/~shyam))