CREATION, INTEGRATING AND DEPLOYMENT OF DIAGNOSER FOR WEB SERVICES

by

MOHAMMED IBRAHIM ALODIB

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
April 2011

ABSTRACT

Service oriented Architecture (SoA) is a layered architecture for organising software resources as services, so that they can be deployed, discovered and combined to produce new Web services. One of the key challenges of SoA is in identifying the occurrence of failures that may result in violations of Service Level Agreements, causing financial penalties or customer dissatisfaction. Therefore, it is crucial to develop methods of on-line detection of failure to take suitable remedial actions without delay. One of the methods of identifying occurrences of failure is to use Diagnosers; software modules which are deployed with the system to monitor the interaction between the services. This thesis presents a diagnostic approach for SoA based on extending the Diagnosability theory of Discrete Event System (DES). In particular, this research has resulted in a method of automated creation of Diagnosers and integrating them to the system. This is accomplished by coming up with an appropriate modelling language framework, which is a prerequisite to applying DES techniques. Modelling languages popular in DES, such as Petri nets and Automata, despite being sufficiently adequate for modelling, are not well adopted by the SoA community. Inspired by Petri nets and Workflow Graphs, a modelling approach, which closely follows BPEL, is proposed. Then, one of existing DES methods is extended for the creation of centralised Diagnoser. Various methods are proposed to implement and integrate the produced Diagnoser into the system. As a proof of concept, an implementation of the suggested approach is created as a Plugin for Oracle JDeveloper. A series of empirical results on the performance-related aspects of the proposed method are discussed.

To my parents,

my loving wife,

my treasured sisters and brothers,

and my cherished nephew and nieces.

ACKNOWLEDGEMENTS

Firstly, I would like to thank all the people who have encouraged and helped me during my PhD project, and without whom this thesis might not have been written. I would especially like to thank my supervisor, Dr Behzad Bordbar, for his encouragement, advice and constant source of inspiration and motivation all these years. I would also like to thank the members of my thesis group for their comments and interest in my research. I would also like to thank the staff of BT research laboratory at Ipswich in particular Basim Majeed and Xiaofeng Du for their fruitful collaboration.

Warm thanks go to my office mates: Peter Lewis, Seyyed Shah and Nur Hana Samsudin. I am also very grateful to my beloved friends in the lovely city of Birmingham.

I would like to thank the government of Saudi Arabia for funding my studentship. Also my thanks go to the School of Computer Science, University of Birmingham for providing such an inspiring environment for me to complete my research.

Finally, my thanks go to all the members of my family: to my Mom and my Dad for their steadfast support, which was greatly needed and deeply appreciated; to my wife for always being there supporting me, I am greatly indebted to her.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Computer network technologies have improved rapidly since the first computer network appeared in the early 1990s. Modern network-based computer systems first began with the evolution of distributed computing techniques. Distributed computing is the allocation of components at a networked computer in order to communicate and coordinate their action by passing messages [45]. The main purpose of a distributed system is to share resources between the system's components. The Internet or World Wide Web [129, 24] is considered the most successful computing network for distributed systems. However, developing and designing systems based on such networks involves some challenges. One of the most significant challenges is how to solve the heterogeneity and interoperability issues: for example, programs written for different development environments and platforms may not communicate successfully with other programs unless they use common standards.

The traditional distributed systems have been improved to overcome such challenges and to provide effective communications, and a set of platforms, such as CORBA [98], DCOM [120, 59], and Java RMI [46], have been proposed. Such platforms aim to enable users to access services and run applications over heterogeneous computers and networks. However, such platforms themselves have another heterogeneous problem as some of them are platform dependent, while others are programming languages dependent: for example, Java RMI supports only

a single programming language (i.e. applications interacting with each other with the help of RMI must be programmed by Java). As another example, developing applications using DCOM requires allocating the applications that communicate with each other at the Windows operating system and using VB, C, C++, or C# to programme of the system. As a result, CORBA has been proposed to solve such heterogeneity issues facing RMI and DCOM platforms. The primary goal of CORBA is to allow different application programs to communicate with one another irrespective of their programming language, operating system and hardware platforms. However, CORBA has not been widely adopted. This could be because it attempts to establish new communication protocols such as IsIOP, instead of using existing TCP/IP standards such as HTTP, SMTP, and FTP, which are popular because of the widespread use of Internet and Web [111]. In addition, the previous platforms face another issue related to security threats caused by using binary messages and specific ports for the communication protocols. These drawbacks have led to a lack of heterogeneity for developing integrated and distributed systems [45]. In order to solve such issues, Service oriented Architecture (SoA) was born.

1.1 Service oriented Architecture (SoA)

Service oriented Architecture (SoA) is an evolution of the previous platforms designed to provide a layered architecture for organising software resources as services, so that they can be deployed, discovered and combined to produce new Web services [73]. Web services provide a favourable solution for solving the problem of integration among autonomous and heterogeneous software systems [87]. They are based on providing a set of interoperable services that are well-defined, self-contained, and do not rely on other services. Unlike traditional distributed system platforms, Web services make use of existing standards as fundamental building blocks. A set of XML-based standards such as WSDL (Web services Description Language) [94], SOAP (Simple Object Access Protocol) [32, 108], and UDDI (Universal Description, Discovery and Integration) [97], are used to encapsulate data, describe the Web services interfaces,

exchanging data and deploying the Web services into the server. Most of these protocol standards are widely accepted by the industry.

In general, SoA is a prevailing software engineering production, which ends the domination of the traditional, distributed system platforms [91]. The growth rate of SoA use in industry has been estimated to be over 24% from 2006 to 2011 [63], and the rapid move towards SoA has been encouraged by the positive results already recorded; for example, the level of reusability in SoA has, on average, been enhanced to more than 2.5 times that of non-SoA development [112]. The cost saved in software development for 2006 to 2010 is reported as an aggregate $50 billion in firms worldwide [84, 67]. In general, the benefits of SoA that organisations can receive are as follows [25]:

- Saving money, time and effort over the long terms through reuse of "components" because of the flexibility of SoA.
- Eliminating frustrations with IT through flexible solutions and shorter lead times to deployment.
- Justifying IT investments more transparently through the closer association of IT to business services.
- Providing business executives with a clear understanding of what IT does, and its value.

In the past few years, SoA has been adopted and widely used by the IT industry. One of the most popular standards of such adaptations is Business Process Execution Language for web services (BPEL) [26]. BPEL is a modelling language used to specify a sequence of actions within business processes in order to build enterprise applications. BPEL offers a rich number of diagrammatic notations ideal for supporting the modelling of complex behaviours such as sequential, parallel, iterative and conditional.

1.2 Fault Detection in SoA

The ability to deliver reliable and dependable enterprise applications based on business processes of higher quality within budget continues to challenge most IT organisations, such as Electronic Banking, Telecommunication, Airlines and Medical systems. A business process in such systems can potentially perform crucial functions upon which the users are heavily dependent. Any type of breakdown often has consequences in terms of customer dissatisfaction, and may result in a violation of a Service Level Agreement (SLA), resulting in fines. For example, any breakdown in a Telecommunication service may result in penalties. As a result, it is necessary to maintain the reliability and dependability by identifying the occurrence of failures accurately, so that suitable remedial actions can be adopted to allow an enterprise to closely monitor their business processes, improve services to satisfy new market needs, and quickly identify and recover from any process failures.

SoA-based systems can fail because of the failure of the underlying services or hardware resources. Another source of failure is through the execution of an undesirable sequence of actions. Such cases are often modelled as part of the business process, but represent cases where the execution of events produces an undesirable result. This type of failure is related to the underlying business process that governs the interaction of the service. A failure caused by a breakdown of a service is often dealt with the software Exception Handling, whereas detection of failure caused by wrong execution of a business process often requires the provision of additional infrastructure to monitor the service interactions. For example, Right-First-Time failure is a category failure which telecommunication companies wish to avoid. This type of failure occurs when a business process fails to complete a task correctly first time, therefore it is forced to repeat a part of the task or the entire task again. When a task is executed more than once, it indicates an incorrect execution of the tasks the first time, or invocation of an erroneous execution scenario. If such failures are identified, firstly, it is possible that a suitable course

of action be adopted to minimise the effect of the failure; secondly, it is possible for business process designers to create strategic solutions to prevent the same failure reoccurring in the future. For example, if human errors are the main cause of a certain failure, then additional training can be introduced. If the failure is caused by poor design, then process designers can consider redesigning the problematic process. If the failure is due to the limitation of the hardware or software capability, then upgrades and the allocation of additional resources can be arranged.

Recently, monitoring and diagnosis applications developed for SoA have been the subject of considerable research [136, 137, 38, 20, 116, 40, 123, 48, 125, 95]. Proposed methods provide run-time quality assessments for monitoring data or recovering from faults caused by the Exceptions. In general, the current process monitoring technologies are mostly based on a system log, which means that when a failure is identified, it has already happened. To prevent delivering wrong or faulty services to customers, failure should ideally be identified and recovered before the execution of a process ends. In this case, a real-time or near-real-time process monitoring technique is essential. Almost all the current BPEL execution engines have real-time BPEL execution monitoring functionality. However, these monitoring methods are mainly provided to BPEL developers for the purpose of debugging, and cannot for example identify and diagnose failures caused by undesirable scenarios such as Right-First Time (RFT) Failures.

Researchers in the Discrete Event System (DES) community have been dealing with similar challenges, mostly aimed at embedded systems, for the past two decades. One of the most successful methods of monitoring systems in DES is to use Diagnosers [118]; software modules which are deployed with the system to monitor the interaction between the system components for identifying if a failure has happened or may have happened. In other words, Diagnosers should answer the following questions: “Did a fault happen or not?” (Online Detection), “What type of fault happened?” (Isolation of Failure) and “How did the fault happen?” (Explanations).

The Diagnoser provides the system diagnosis based on its (complete) knowledge of the over-

all system model and the overall system observation of the events. In this context, the occurrences of *observable* events are reported to the Diagnoser, while the set of *unobservable* events are internal actions that occur silently in the system, which means their occurrences remain hidden from the Diagnoser and outside world. Obviously, the set of failure events is classified as unobservable events, otherwise their detection would be trivial. Therefore, the Diagnoser uses the system model and only the observable actions in order to infer the system status and to answer the above questions about online detection, isolation of failure and explanation.

A variety of DES algorithms to compute the Diagnoser for embedded computing systems has been proposed [118, 55, 71, 69, 117, 57]. Such algorithms have been adopted and successfully applied to various technological areas e.g., telecommunication networks [23, 49], power systems [21, 70, 61] and production systems [31, 90, 139]. Based on the success achieved by adopting the DES algorithms to such systems, this thesis aims to exploit and adopt such algorithms in SoA.

Methods suggested by the DES community for the design of the Diagnoser are mostly reliant on representations such as Automata [118] or Petri nets [55, 71]. Adopting such methods for SoA requires transforming models of the system, which are often captured at higher levels of abstraction, e.g. BPEL, into lower level models in Automaton [137, 136, 10, 8]; this requires a substantial bridging of the gaps between different types of modelling languages. This is because Automata and Petri nets, despite being sufficiently adequate for modelling, are not well adopted by the SoA community. Therefore, coming up with an appropriate modelling language framework is a prerequisite to applying DES techniques.

Inspired by Petri nets and Workflow Graph, a modelling approach based on adopting and extending the conventional Workflow Graph [128], which closely follows BPEL standard, is introduced as a modelling language for specifying models of business processes. Then, we adopt the Diagnosability of DES [118, 34, 57, 58] to the presented language. This results in a formal foundation for the automated creation of the Diagnoser for SoA.

This thesis goes further by proposing various methods to automatically integrate and deploy the produced Diagnoser into the system, so that the Diagnoser can be deployed with the rest of the system. The integration of the Diagnoser has been automated with the help of the Model Driven Architecture (MDA) [99]. Using MDA promotes the role of modelling and the automated creation of models and codes to bridge the gap between technical spaces. As a proof of concept, the presented approach has been implemented as a Plugin for Oracle JDeveloper. We have evaluated our approach by using a simplified case study for resolving faults in Customer broadband connect. This case study is provided by our industrial partner British Telecom, BT. An overview of the proposed approach is presented in the next section.

1.3 An Overview of our Approach

In this research, we consider the development of the business process from the perspective of fault diagnosis. Figure 1.1 depicts an outline of our approach to create a method of detection of failures in SoA. In particular, this method aims to produce a diagnosing service which is automatically integrated with the existing services of the system in order to identify the occurrence of failures. The creation of this service is based on adopting and extending the well-established algorithms of Discrete Event System (DES).

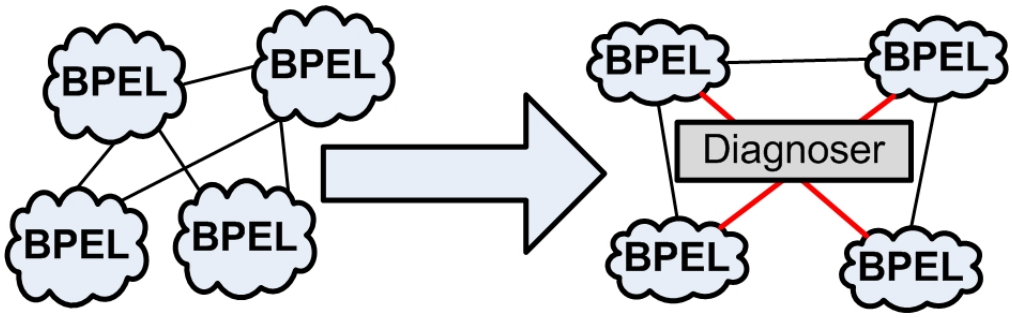

Figure 1.1: Diagnosing Service

To achieve this approach, we have adopted the conventional Workflow Graph [128] to capture business process models. This modelling language is suitable for our approach for the

following reasons: firstly, Workflow Graph is a rich modelling language. It includes essential constructs, such as Fork, Join, Merge and Decision, which are commonly used in the modelling of business processes. Secondly, Workflow Graphs have strong semantics based on Petri nets, which allows formal approaches to formulating failure diagnosis. Finally, to the best of our knowledge, the formalism of [128] is the closest to the standards, such as Business Process Execution Language (BPEL), which are widely used by the industry and tool vendors e.g., Oracle JDeveloper [92] and IBM WebSphere [65].

The conventional Workflow Graph [128] includes conditional nodes which allow expressing repetitive behaviours by creating a loop in the model. However, creating loops by making cycles may lead to *unstructured loops* [39, 43], e.g., loops with multiple entries. Such loops may result in the elimination of the parallel behaviour producing inefficient code [13]. Leading tool vendors such as IBM WebSphere and Oracle JDeveloper do not support the creation of unstructured loops. Instead, they introduce high-level constructs called "While loops" [50, 79, 73]. As a result, to develop a theory for the realistic modelling of such repetitive behaviours we have extended the adopted model of [128] by providing support for *While* loop as specified in the BPEL standard. Our extension of the Workflow Graph, which is called the *Extended Workflow Graph (EWFG)*, results in a tree of Workflow Graphs where a While activity is a node with a (unique) child, which is also an Extended Workflow Graph, representing the behaviour that should be repeated when the While occurs. We have also extended the conventional Workflow Graph [128] to create formalism for supporting *Invocation* activity of BPEL, which are widely used in business process models.

The presented formalism is used to prove that models created from the widely used subset of BPEL produce regular languages. In other words, the language underlying models supported by these tools, such as BPEL, is a regular language. As a result, the Diagnosability theory of DES can be adapted to the Extended Workflow Graph, as the Diagnosability theory of DES requires a regular and finite model to be applied.

A listing of the contributions of this thesis is presented in the following section.

1.4 Contributions of the Thesis

The contributions of my research are outlined as follows:

- Enhancing models to include information about Observability and failures are introduced (Chapter 3).
- A modelling language for the Business Process based on extending the conventional Workflow Graph is introduced. It aims to raise the level of abstraction to create a modelling framework close to BPEL. More specifically:
    1. Extending the conventional Workflow Graph to avoid *Unstructured loops* by introducing the notation of *While* loops and the *Invocation* node (Section 4.4).
    2. Introducing a semantic for the *Extended Workflow Graph* (Section 4.4.2).
- A model-based diagnosis approach to produce the Diagnoser is presented. This includes extending the Diagnosability theory of DES [118] to the Extended Workflow Graph. In particular:
    1. Enhancing the Extended Workflow Graph to include information required to deal with the Observability (Section 5.1).
    2. Proposing an algorithm to compute the Coverability Graph (Section 5.3).
    3. Proving that models produced by the Extended Workflow Graph are regular (Section 5.3.1).
    4. Proposing an algorithm to compute the Diagnoser of systems modelled as an Extended Workflow Graph (Section 5.4).
- There are different possible architectures for integrating the produced Diagnoser into the system. More specifically:

1. Proposing two methods to implement the produced Diagnoser in SoA, one is to produce the Diagnoser as a BPEL service and the other to produce it as a Web service (Section 6.1).
2. Proposing two methods to integrate the implemented Diagnoser into the system:
   - By adding extra invocations to execute the Diagnoser (Section 6.2.1).
   - By using a *Protocol Service* model to accomplish the interaction between the system services and the Diagnoser (Section 6.2.2).
3. Introducing MDA model transformations to automate the implementation and the integration of the Diagnoser.

- As a proof of concept, the approach presented in this thesis is implemented as a Plugin for Oracle JDeveloper (Chapter 7).
- A case study provided by our industrial partner BT is used to evaluate and demonstrate the feasibility of the presented approach. This case study involves a scenario to resolve broadband problems (Chapter 8). The case study is also used to evaluate the integration methods of the Diagnoser from a performance point of view.

1.5 Publications

The outcome of this research has been disseminated in the following seven papers. This thesis should be regarded as the definitive account of the work.

- *Mohammed Alodib*, Behzad Bordbar and Basim Majeed. **A model driven approach to the design and implementing of fault tolerant Service Oriented Architectures**. In the IEEE International Conference on Digital Information Management (ICDIM), London, pp. 464-469, 2008.

- *Mohammed Alodib* and Behzad Bordbar. **A Model Driven Architecture approach to fault tolerance in Service Oriented Architectures, a performance study**. In the IEEE Enterprise Distributed Object Computing Conference Workshops (EDOCW), Munich, Germany, pp. 293–300, 2008.

- *Mohammed Alodib* and Behzad Bordbar. **A Model-Based Approach to Fault Diagnosis in Service Oriented Architectures**. In the IEEE European Conference on Web Services (ECOWS), Eindhoven, The Netherlands, pp 129-138, 2008.

- *Mohammed Alodib* and Behzad Bordbar. **On automated generation of Diagnosers in Fault tolerant Service oriented Architectures**. Journal of Digital Information Management, 344-350, Volume 7 Issue 6, 2009.

- *Mohammed Alodib* and Behzad Bordbar. **A Modelling Approach to Service oriented Architecture for On-line Diagnosis**. Submitted as a journal paper to IEEE Transaction on Service Computing (TSC).

- *Mohammed Alodib*, Behzad Bordbar, Xiaofeng Du and Basim Majeed. **On Diagnosis of Failures in Business Processes Involving Iterations**. Submitted for publication.

- Xiaofeng Du, Behzad Bordbar, *Mohammed Alodib* and Basim Majeed. **Applying Protocol Service for the Monitoring of Business Process**. In the IEEE GCC Conference and Exhibition, pp 633–636, 2011.

1.6 Thesis Outline

The structure of the thesis is as follows:

- Chapter 2 introduces the reader to preliminary concepts, such as Service oriented Architecture (SoA), Oracle Fusion Middleware, Diagnosability of Discrete Event System

(DES), Petri nets, Workflow Graph and the Model Driven Architecture (MDA). Moreover, a review of existing literature on diagnosing failure for Service oriented Architecture (SoA) is discussed.

- Chapter 3 describes the annotations requires to adopt the DES techniques. This includes annotating the BPEL activities with new information related to the observability and failures.
- Chapter 4 presents a modelling approach called the Extended Workflow Graph which is proposed to specify models of business processes.
- Chapter 5 introduces a model-based approach which is based on adopting the Diagnosability of DES to the Extended Workflow Graph.
- Chapter 6 presents various methods to integrate the produced Diagnoser into the system.
- Chapter 7 describes the implementation of the presented approach.
- Chapter 8 demonstrates the feasibility of the presented approach by using a simplified version of a case study provided by BT.
- Chapter 9 summarises our work and points out the potential future directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter presents an introduction to the concepts, which will be used throughout this thesis. Firstly, an introduction to the fundamentals of Service oriented Architecture (SoA) is described. This includes describing Web Services, Business Process Execution Language (BPEL) and Web service integration. Then, the Oracle Fusion Middleware framework will be explained. Next, the concept of the Discrete Event Systems (DES) theory will be presented, and Petri nets and Workflow Graph are introduced. Finally, the principles of Model Driven Architecture (MDA) will be presented.

2.1 Services oriented Architecture (SoA)

Component-based systems have been developed upon the traditional software architecture for the past 40 years. Recently, building distributed components over the Middleware in order to provide integrated systems and processes have become a key demand of the IT industry. To meet this demand, a set of platforms, such as CORBA [98], DCOM [120, 59], and Java RMI [46], have been proposed. However, such platforms are not fully interoperable, as some of theme are platform dependent, while others are programming languages dependent. For example, Java RMI supports only a single programming language (i.e. applications interacting with each other with the help of RMI must be programmed by Java). In this case, only en-

terprise applications using the same communication technology can be directly integrated. In addition, these platforms have a security issue, as the communication protocols make use of binary messages and specific ports, which may not be permitted by the firewall. These drawbacks have led to a lack of heterogeneity and interoperability for developing integrated and distributed systems [45]. Therefore, Services oriented Architecture (SoA) was introduced to address such challenges.

SoA is a framework which provides a layered architecture for organising software resources as services, so that they can be deployed, discovered and combined to produce new services [73]. A simple SoA infrastructure involves three independent collaborative components which are described as follows [110, 81], see Figure 2.1:

- **Service Provider**: The service provider is responsible for publishing the services. In other words, the service provider is the owner of the services, such as companies and organisations.

- **Service Requester**: A requester is a client or organisation that wishes to make use of a provided service. The requester searches for the desired Web services in the service registry.

- **Service registry**: A global registry acts as a central service which provides a directory where service descriptions are published by the Service Provider. Then, *Service Requesters* find service descriptions in the registry and obtain binding information for services from the *Service Provider*.

2.1.1 Web Services

The W3C organisation defines the fundamental architecture of the Web services technology. The goal of the Web services architecture is to provide a promising solution for the problem of integration among autonomous and heterogeneous software systems [87]. In a sense, they

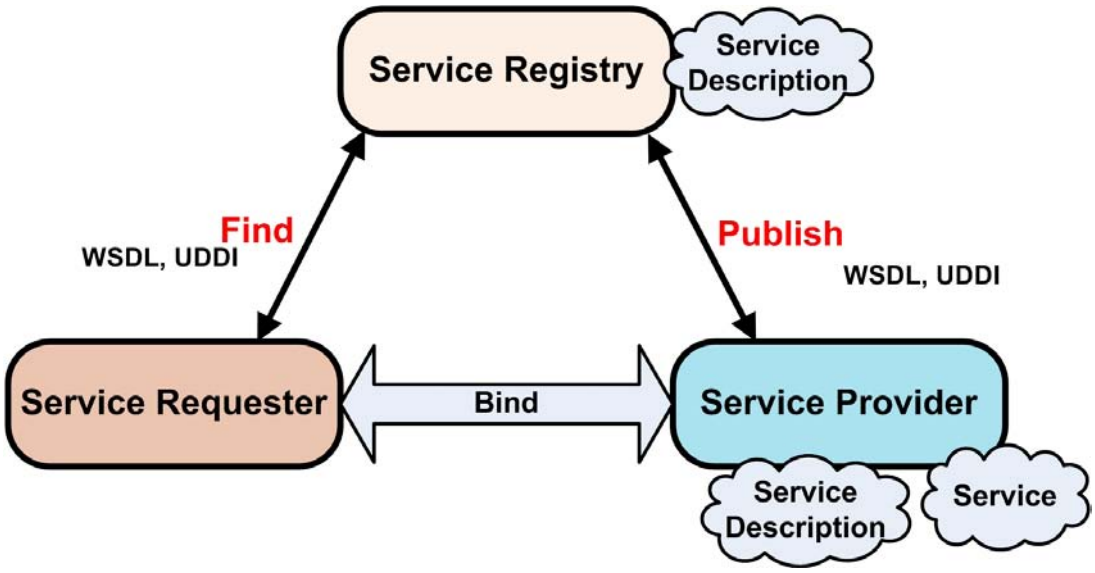

Figure 2.1: A Basic Service Oriented Architecture [81]

extend the Web from a distributed source of information to a distributed source of services. In particular, Web services provide specific properties and capabilities, which enhance the development of software. The most important features of Web services are described as follows [37, 52]:

- Modular: Web services are considered as software components, self-contained, reusable, possibly forming into larger components.
- Implementation-Independent: Services are offered in a way that is independent of the final implementation.
- Available: Services are available to other systems, so they need to be exposed and published.
- Published: Service descriptions are made available in a repository, so those requesting them can find the service and use the description to access the service.
- Interface Based: Services are exposed and defined via accessible interfaces.

- Loose Coupling: Web services are based on loosely coupled application components, where each component is exposed as a service. The notion of loose coupling precludes any knowledge or assumptions about the implementation specifics or the formats and protocols used.

- Self Describing: Services have a machine-readable description that can be used to identify the interface of the service and its location.

- Discoverable: Services need to be found not only by a unique identity but also by interface identity and by service kind. Applications do not know in advance where the service is or how to be invoked.

- XML Based: Web services endorse XML as a standard format for structuring data and content for electronic documents.

- Synchronous or Asynchronous: Inherently, Web services are based on asynchronous communications. However, synchronous messaging and remote procedure styles are also supported.

A Web service is a part of the Middleware technology which provides integrated and interoperable interactions over the Internet [28]. As described above, Web services are well-defined, self-contained, loosely coupled, self describing and modular applications that can be published, located and invoked across the Web network [87]. These features give Web services the ability to be dynamically invoked by other applications or other Web services, and composed with other services to achieve complex tasks. In other words, Web services are highly reusable components which act as building blocks to develop service composition, as well as solving the application communication and integration issues.

The development of a composite web service is built upon the Service oriented Architectural paradigm [12]. Communication in a composition of Web services is based on using

well-accepted standards and the XML messaging framework [83]. Such standards are used to encapsulate the service's business logic and functionality in order to expose only the functionality, not the implementations via accessible interfaces. Therefore, application programs communicate with one another irrespective of their programming language, operating system and hardware platforms.

Traditionally distributed computing platforms such as DCOM and CORBA, in comparison with Web services, are based on quite different and incompatible object models in their communication [11]. For example, CORBA makes use of new communication protocols such as IsIOP, which lead to interoperability problems. Web services overcome such issues by using the common Extensible Markup Language (XML), XML Schema Definition XSD [124] and the standard TCP/IP based communication protocols.

Various XML based standards are used by Web services in order to describe their architecture, intercommunication, collaboration and discovery [87]. In particular, the communication messages between a Service Requester and a Service Provider are encoded into *the Simple Object Access Protocol (SOAP)* messages which are plain text XML messages. The *Web Services Description Language (WSDL)* is used to describe the invocation details of a Web service, such as the service name, the available operations, and the information related to the input and output variables. The *Universal Description Discovery and Integration (UDDI)* provides protocols for querying and updating Web service information. For communication purposes, Web services utilise the existing standard TCP/IP protocols such as HTTP, HTTPS, SMTP, and FTP [111].

2.1.2 Service Interaction and Integration

Combining Web services in order to create a collaborative application requires standards to model the interactions. Moving from simple independent invocations to sequences of operations has important implications both internal (implementation) and external (interaction) perspective [12]. From internal perspective, the user must be able to maintain context information and ex-

ternal is to call exposed operations. Web services interaction is considered either orchestration or choreography [11].

Orchestration

The interactions of orchestration architecture are controlled by a single endpoint (coordinator), see Figure 2.2. The coordinator controls how the Web services can be involved and executed. In this architecture, the involved Web services do not know that they are involved in a composition application. Only the central coordinator of the orchestration is aware of this involvement, therefore the orchestration is centralised. In this thesis, the *Protocol Service*, which is introduced in Chapter 6, is represented as orchestration architecture.

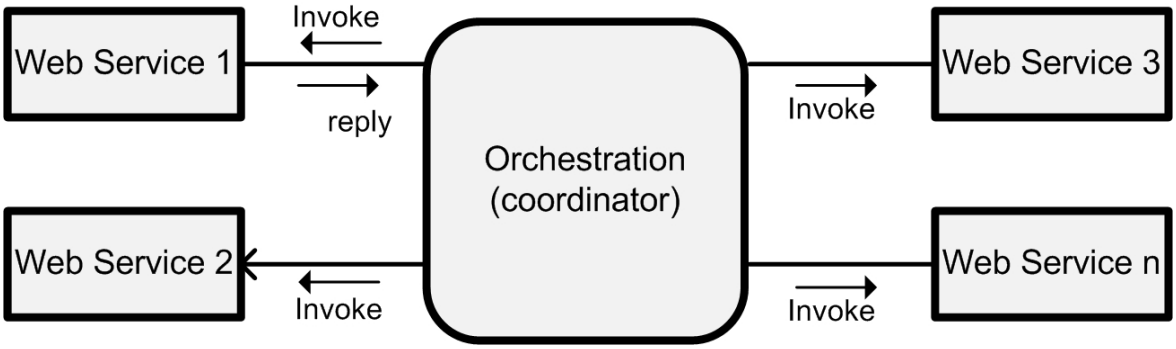

Figure 2.2: Composition of Web services with Orchestration Architecture

Choreography

Choreography architecture, when compared to orchestration architecture, does not rely on a central coordinator as depicted in Figure 2.3. Web services involved in this architecture know exactly when to execute its operations and with whom to interact. Choreography can be seen as a collaborative effort focused on the exchange of messages between business processes. In general, choreography is more collaborative in nature than orchestration as it is described from the perspective of all parties, and explains the behaviour between participants in collaboration.

From the perspective of composing Web services, *orchestration* is a more flexible architec-

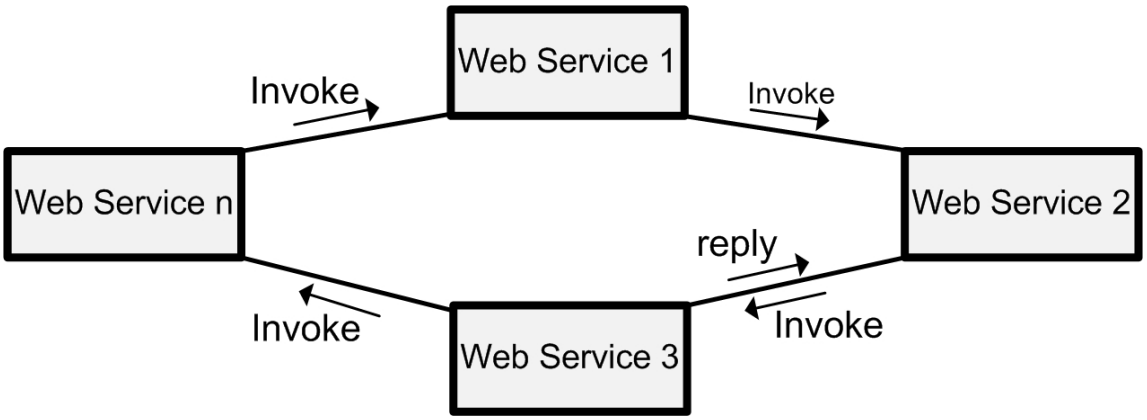

Figure 2.3: Composition of Web services with Choreography Architecture

ture offering the following advantages over choreography [73]:

- The coordination of component processes is centrally managed by a coordinator.
- Web services can be incorporated without being aware that they are taking part in a business process.
- Providing alternative scenarios in case of faults.

In the last few years, a set of XML languages have been proposed as implementations of orchestration and choreography such as BPMN [3], BPEL [73, 22], XPDL [2] and WS-CDL [130]. The focus of this thesis is to use Business Process Execution Language (BPEL).

2.1.3 Business Process Execution Language (BPEL)

Business Process Execution Language for Web Services (BPEL, WS-BPEL, BPEL4WS) is a graphical language used for the composition, orchestration, and coordination of web services [73]. Combining and linking existing Web services and other components to deliver new composition services is called *business processes*; therefore, BPEL is used to specify a set of actions within business processes, in order to achieve a common business goal. The BPEL specification is based on using the Web Services Description Language (WSDL) [37], which is an XML

language describing services as a set of accessible interfaces, for producing business processes supporting interoperability [22].

BPEL offers a rich set of diagrammatic activities ideal for supporting the modelling of complex behaviours such as sequential, parallel, iterative and conditional. In addition, similar to traditional programming languages, BPEL offers constructs, such as loops, branches, variables and assignments. Consequently, such constructs make the definition of complex business processes possible [73]. For further information about Web services and BPEL, please refer to Appendix A.

BPEL is supported by the majority of software vendors such as Oracle, Microsoft, IBM, BEA, SAP, and others. In this thesis, we make use of Oracle Fusion Middleware 11g, which will be explained in the following section.

2.2 Oracle Fusion Middleware 11g

Oracle Fusion Middleware 11g (OFM) is a comprehensive infrastructure of pre-integrated, industry-leading Middleware, for the development, deployment, and management of Service oriented Architecture (SoA) [105, 104]. OFM consists of multiple services, including Java EE and developer tools, integration services, business intelligence, collaboration, and content management. In addition, the composite applications running on the OMF can make use of the following service languages and engines for executing its components [68]:

- BPEL Process Manager Orchestrates (potentially) long-running service composites with many interactions with external services.
- Decision Service or Business Rules engine: executing decision logic that can be (re)defined at run time.
- Human Workflow Service: for engaging humans in making decisions or providing information.

- Mediator: for filtering, transforming, adapting, and routing messages.
- BPMN: business process logic defined through BPMN [3] can be executed inside the OMF (by the same engine that also runs BPEL).

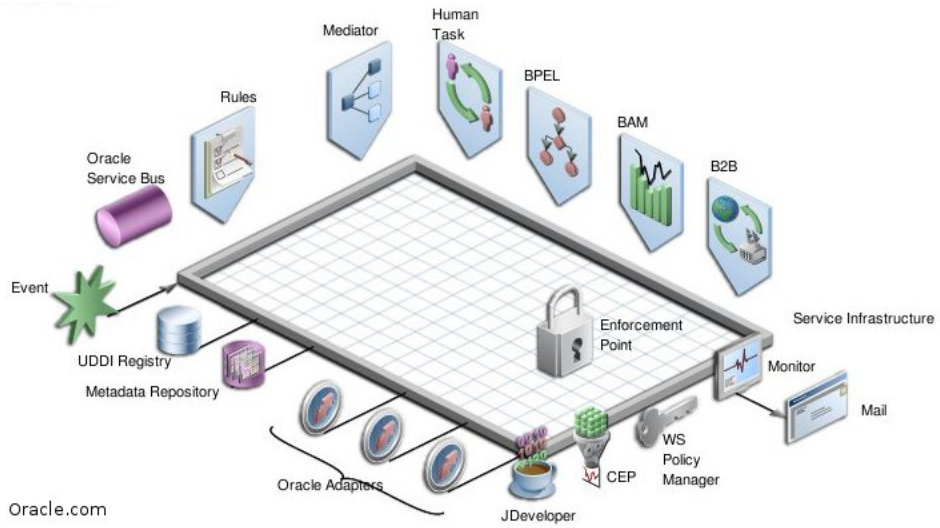

Figure 2.4: Oracle Fusion Middleware Architecture [106]

The OMF is shipped and integrated with a set of technology adaptors, see Figure 2.4. Such adaptors provide specific protocols and languages to external technology platforms. Examples of these protocols, platforms, and languages include the file system, FTP servers, the Database, JMS queues, the eBusiness Suite, SAP, and various Business-to-Business (B2B) exchange types, such as RosettaNet, ebXML, HL7, and EDI (FACT). These enable SoA applications to connect to different components. In particular, MOF composite applications can access the following components [68]:

- Oracle Database: for accessing tables and views (query and data manipulation) and calling PL/SQL program units
- File and FTP: for reading and writing files from a file system and an FTP server.
- Queues: for accessing queues through JMS, Oracle Advanced Queuing, and MQ Series.

- Enterprise Java Bean (EJB): communicating with remote Enterprise JavaBeans.
- Sockets: for reading and writing data over TCP/IP sockets.
- Oracle eBusiness Suite adaptor: for retrieving data from and sending data to eBusiness Suite.
- Business Activity Monitoring (BAM): for sending data and events to an Oracle BAM server, which is a complete solution for building interactive, real-time dashboards and proactive alerts for monitoring business processes and services [107].
- Business-to-Business (B2B): for the exchange of business documents with e-commerce trading partners such as RosettaNet, HL7, and various EDI protocols; and support interaction with SAP and other ERP applications

To develop an enterprise application on Oracle Fusion Middleware 11g, three platforms are required: *Oracle JDeveloper*, *Oracle SoA suite* and *Oracle Weblogic server* [88]. *Oracle JDeveloper* is an *Integrated Development Environment (IDE)* which provides the required tools, such as BPEL, Business Rules and Web Services Manager, for creating composite applications. In particular, the SoA Composite Editor, which is involved in Oracle JDeveloper, is based on drag-and-drop fashion with a variety of components and technologies that support building composite applications. *Oracle SoA suite* is a set of service platform components for building, and managing SoAs. *Oracle Weblogic Server* provides developers with the tools and technologies to deploy and publish enterprise applications and services. For more details on Oracle Fusion Middleware 11g, please refer to [104, 103].

2.3 Discrete Event Systems (DES)

A Discrete Event System (DES) is a *discrete-state*, *event-driven* system whose state depends on the occurrence of asynchronous discrete events over time [36]. It is assumed that events occur

for a reason (i.e. condition), which changes the state of the system from one state to another. The state changes only at discrete points in time through transitions associated with events. The behaviour of a DES system is described in terms of event sequences of the form $e_1 e_2 e_3 \ldots e_n$, which can be modelled by a *language*.

2.3.1 Language Models of Discrete Event System

The theories of languages are considered one of the formal methods to study the logical behaviours of DES [36]. It is known that a system in DES has a set of Events. Such events are considered as the *alphabet* of a *language* and any sequence of these events are *words*. Let us motivate this discussion of languages by considering a simple example. Suppose a simple photocopier machines which can be turned on frequently, and we would like to design a simple system to carry out the following tasks. Firstly, when the machine is turned on, it should issue a signal to tell the user that it is "ON" and waiting for requests. Such a state gives the user a simple status report. Each signal defines an event, and hence the set of possible signals produced by the machine represent an alphabet of the system. The DES in this model has the responsibility to check events in order to give the right explanation. For example, the event sequence: "ON", "everything is OK", "status report done" means that our task has been successfully completed, while "ON", "status report done" with no action in between may explain that as an abnormal behaviour. In addition, if the status "ON" is seen but the execution does not end in status "status report done", it indicates that something goes wrong. As a result, such combinations of signals issued by the machine are thought as words associated to the language of this system.

To build a language for any system, the set of events Σ is viewed as a set of alphabet. This set is assumed to be finite. Any sequence produced from the combination of these events is called a "trace" (or "word"). If we have a trace which does not have any event, it is called the *empty trace* and denoted by ε. The length of a trace is the number of the events included in it.

Definition 2.3.1 (language [36]) *A language defined over an event set Σ is a set of finite-length*

strings formed from events in Σ.

Example 2.3.2 *Let* $\Sigma = \{a, b, c, d\}$ *be the set of events. We may define the language* $L_1 = \{\varepsilon, a, abc, abb\}$*, which has only four strings, or the language* $L_2 = \{$*all possible strings of length 4 start with event b*$\}$.

It is noteworthy that the language of the system L is a subset of Σ^*, where Σ^* denotes the *kleene-closure* which is the set of all finite strings of elements of Σ, including the empty string ε [89]. Therefore, Σ^* is countably infinite as it has strings of arbitrary long length. For example, assume $\Sigma = \{a, b, c, d\}$, then the kleene-closure of Σ is

$$\Sigma^* = \{\varepsilon, a, b, c, d, aa, bb, cc, ab, ba, aab, abbb, \dots\}$$

It can be seen that building traces, and therefore languages, is based on the concatenation of the set of events. For example, the trace abc in L_1 is the concatenation of the trace ab with the event c; while ab is itself the concatenation of a and b. Therefore, the language can be seen as a formal way of describing the behaviour of systems in order to declare all possible sequence of events that the system can execute. For example, the language L_1 explained in example 2.3.2, can be defined by a simple enumeration as it has only four traces. However, it is difficult to represent all possible cases for some languages. For example, it is not easy to define all possible traces of the language L_2 as full enumeration is not possible. As a result, it is necessary to find a way to define languages, where all possible traces can be manipulated. This is achieved by using the modelling formalism of Automata or Petri net as a framework for representing and manipulating languages. Section 2.3.2 describes the fundamental of the Automaton, while the concepts of Petri nets will be explained in Section 2.4.

2.3.2 Automata

An automaton is described as a tool which represents a language according to well-defined rules [36]. In this research we focus on the Deterministic Automaton.

Definition 2.3.3 *A Deterministic Automaton [36], denoted by G, is a six-tuple:*

$$G = (X, \Sigma, f, \Gamma, x_0, X_m)$$

where:

X *is the set of states.*

Σ *is the finite set of events associated with the transitions in* G.

$f : X \times \Sigma \to X$ *is the transition function:* $f(x, e) = y$ *represents that there is a transition called* e *which links state* x *to state* y.

$\Gamma : X \to 2^{\Sigma}$ *is the active event function (feasible event function);* $\Gamma(x)$ *is the set of all events for which* $f(x, e)$ *is defined and it is called the active event set (feasible event set) of* G *at* x.

x_0 *is the initial state.*

$X_m \subseteq X$ *is the set of marked states.*

An Automaton G defines a dynamical system with language L as follows. The model starts from the initial state x_0. The occurrence of an event $e_1 \in \Gamma(x_0) \subseteq \Sigma$ moves the system status from the initial state x_0 to state x_1. Then the process continues, executing a transitions each time for which f is defined.

Inspecting the state transition diagram of the Automaton can show the connection between languages and Automata [36]. Starting at the initial state, all possible directed paths, which can be followed in the state transition diagram, lead to the notation of the *language generated* by Automaton.

Definition 2.3.4 *The language generated by* $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ *is*

$$L(G) = \{s \in \Sigma^* : f(x_0, s) \text{ is defined}\}$$

Example 2.3.5 *Consider the Automaton* $G = (X, \Sigma, f, \Gamma, x_0, X_m)$ *depicted in Figure 2.5 where:* $X = \{x_0, x_1, x_2, x_3\}$, $\Sigma = \{a, b, c\}$. *Then, the language generated by the Automaton is* $L(G) = \{(bab)^* ac^*\}$, *where* $*$ *means Zero or any number (i.e. the range over the natural*

numbers), such that we can have non or any number of bab *followed by* a*, then non or any number of* c*. For instance, the traces* a*,*$baba$*,*ac*,*$babbaba$*,* $babaccc$ *respect the language* $L(G)$*.*

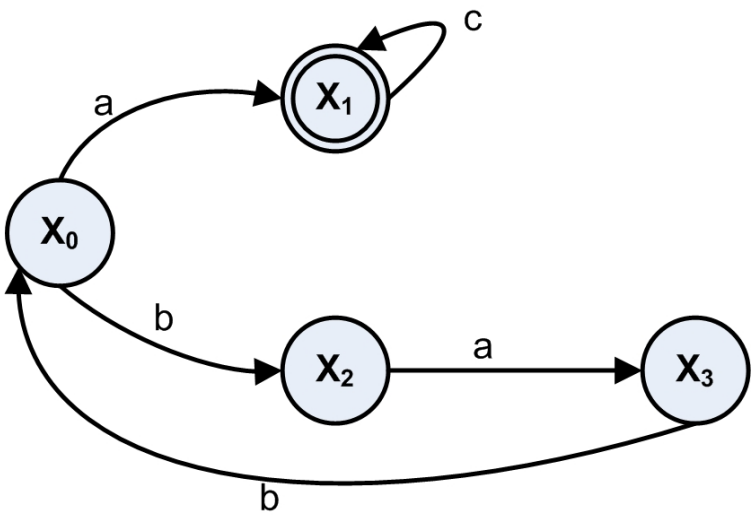

Figure 2.5: State Transition Diagram of Example 2.3.5

2.3.3 Regular Language

As explained in Section 2.3.2, an automaton can be used to manipulate the language of the system and is assumed to represent all possible traces of the language. However, in some cases when the language of the system produces infinite traces, it is impossible to be represented in a finite automaton. A classical example of infinite behaviour is the language $L = \{\varepsilon, \sigma\beta, \sigma\sigma\beta\beta, \dots\} = \{\sigma^n\beta^n : n \geqslant 0\}$ [64]. It can be seen that in this example, a marked state can be reached after exactly executing the same number of β events as that of σ events in the beginning of the trace, as depicted in Figure 2.6. This leads to the following definition of regular languages.

Definition 2.3.6 (Regular language [36]) *A language is regular if it can be marked by a finite-state automaton*

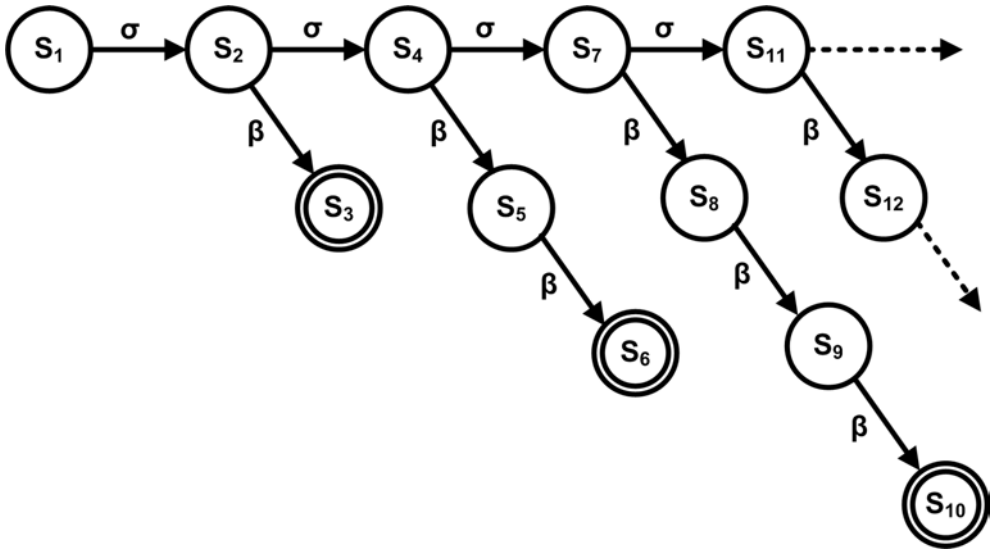

Figure 2.6: Automaton marking the language $L = \{\sigma^n \beta^n : n \geqslant 0\}$.

2.3.4 Observability of DES

From DES perspective, the behaviour of the system is modelled as an Automaton. Some events of the Automaton are considered *observable*, for example, the output of sensors where their occurrences can be seen by other components, while some events are considered *unobservable*, where their occurrences remain silently in the system. Such unobservable events can be due the absence of a sensor to record the occurrence of the event or the event occurs at a remote location. As a result, the set of events are partitioned as $\Sigma = \Sigma_{obs} \cup \Sigma_{uo}$ where Σ_{obs} represents the set of observable events and Σ_{uo} represents the set of unobservable events [118].

From an external service point of view, only the observable events Σ_{obs} can be recognised. If we suppose that an Automaton G executes a sequence of events $\sigma = e_1 \dots e_r$. A *Projection* map would often be used to erase unobservable actions from σ to create the set of observable actions [36].

Definition 2.3.7 *Suppose* $P : \Sigma \to \Sigma_{obs} \cup \{\varepsilon\}$ *is defined by*

$$P(\sigma) = \begin{cases} \varepsilon & \textit{if } \sigma \in \Sigma_{uo} \\ \sigma & otherwise \end{cases}$$

where ε is the identifier of the alphabet Σ (i.e. for $\sigma \in \Sigma$, $\sigma\varepsilon = \varepsilon\sigma = \sigma$). In addition assume extending $P : \Sigma^* \to (\Sigma \cup \{\varepsilon\})^*$ by defining for $P(\sigma_1 \dots \sigma_r) = P(\sigma_1) \dots P(\sigma_r)$ representing the sequence of observable events in $\sigma_1 \dots \sigma_r$ in their right order.

As explained in Section 1.3 that in this thesis we aim to adopt the DES techniques to SoA. Therefore, corresponding to the above concepts, Business process activities, which are explained in Section 2.1.3, can be partitioned into observable and unobservable event in a similar manner, as we will explain in Chapter 3

Modelling of Failures

A failure in a dynamical system is a deviation of the system structure or the system behaviour from the normal situation [82]. For example, the system can fail due to the blocking of an actuator, the loss of the sensor or the disconnection of a system component. Such failures are explicitly modelled by a subset of the unobservable events and denoted by Σ_f. So without any loss of generality we assume that $\Sigma_f \subseteq \Sigma_{uo}$, since observable events can be trivially diagnosed. The set of failures is partitioned into disjoint set corresponding to different failures types: $\Sigma_f = \{F_1 \cup \dots F_m\}$. In the following, we shall introduce a real example which has some observable, unobservable and failure events.

Example 2.3.8 *A common example of a DES system is the Heating System which involves a valve, pump and controller. In addition to these components, there are two sensors: a valve flow and a pump pressure sensor. Each of these sensors has two possible values: Flow (F) or No Flow for the valve sensor, and Positive Pressure (PP) or No Pressure (NP) for the valve pump sensor. In the valve model, see Figure 2.7, the events which change the state of the system*

to SC and SO (i.e. stuck closed and stuck open) are considered unobservable events. Moreover, they represent failure events. In a normal mode, the controller issues the command open valve when it senses a heating load on the system, while the command close valve is issued when the load is removed. Suppose that when the controller fails, it does not sense the load of the system. Thus, the controller does not issue any commands to the valve, although, the system can execute any arbitrary sequence of events that does not include any of the valve commands.

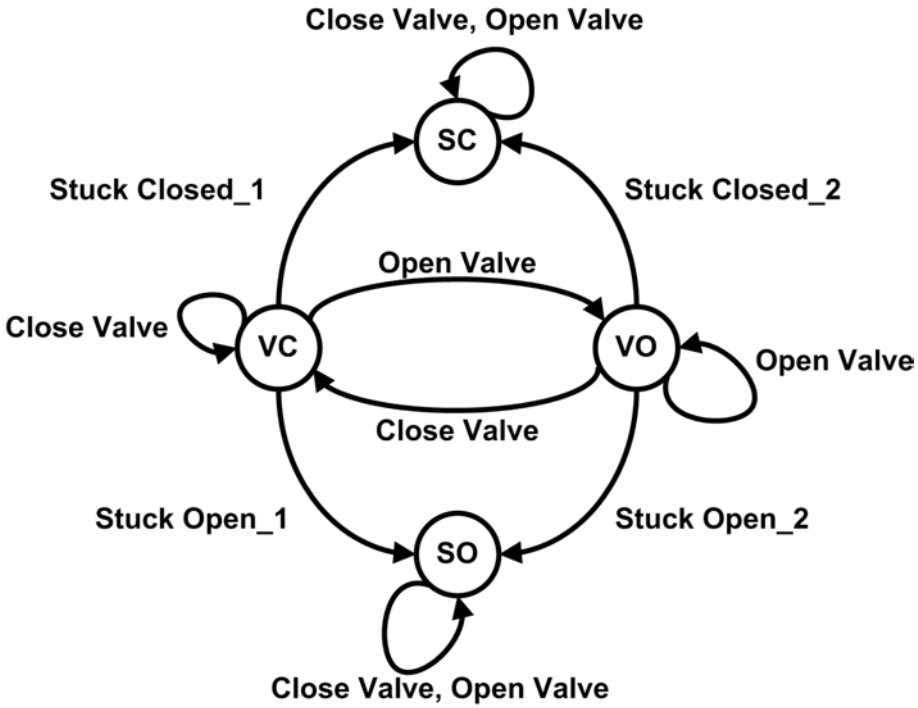

Figure 2.7: The valve model.

In my research we deal with similar failure which can be modelled as subset of the events, as will be discussed in Chapter 3. Other types of failures such as emerging failures which are unknown at the time of modelling fall out of the scope of this research.

2.3.5 The Diagnosability of DES

As explained in Section 2.3.2, the DES system is modelled as an automaton where the set of events are disjointed into two categories: observable and unobservable, as explained in Section

2.3.4. In addition, the failures events are modelled as a subset of the unobservable event set, as explained in Section 2.3.4. It is noteworthy that during the observation of an event, the state of the system is not known exactly due to the presence of unobservable events. Therefore, if these unobservable events represent failures, then detecting their occurrences are very important.

The diagnosis of such failures for discrete-event systems has received considerable attention in the last decade. One of the common methods of identifying failures is by creating Diagnosers, i.e software modules which monitor the interaction between the components to identify if a failure has (or may have) happened. A variety of DES algorithms to compute the Diagnoser for embedded computing systems has been proposed [118, 55, 71, 69, 117, 57]. The general idea behind these methods is based on the use of the knowledge of the overall system model and the overall system observation of the events in order to infer the system state, as shown in Figure 2.8. This can be explained with the help of a simple example.

Figure 2.8: An overview of the Diagnoser Service

Example 2.3.9 *Consider the automaton G shown in Figure 2.9, where the set of events of this model is $\Sigma = \{\alpha, \beta, \lambda, \delta, a, \sigma_{f1}\}$. The set of observable events is $\Sigma_{obs} = \{\alpha, \beta, \lambda, \delta, a\}$, and σ_{f1}, the event to be diagnosed, is the only unobservable event in the system. It can be seen that executing the trace $\alpha\beta\lambda a$ leads to the state 6 where we can infer that the failure σ_{f1} has definitely happened. However, after executing the trace $\alpha\beta\lambda\delta$, there are two possible states. The first is the state 3 where the failure σ_{f1} has occurred. The second is the state 7 where there is no failures occurred (i.e. normal state).*

From this example, it can be seen that from some observable events the status of the system can be inferred. This can be formalised in the following definition.

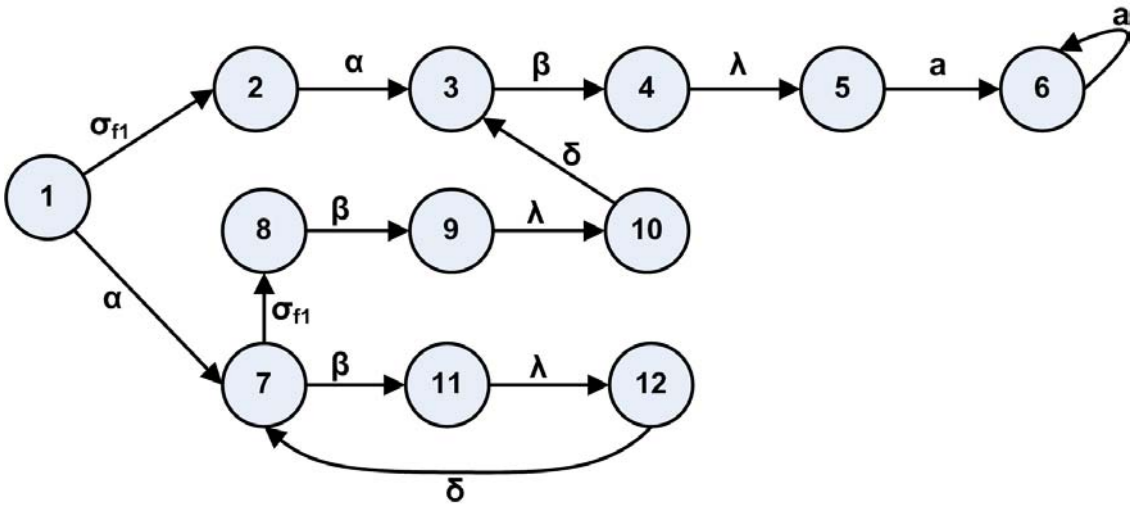

Figure 2.9: Automaton G of Example 2.3.9 [118]

Definition 2.3.10 *Consider an Automaton G with a set of events Σ. Also assume the set of failures are divided into categories $f_1, f_2, \ldots, f_\ell$. Suppose that $\sigma = e_1 \ldots e_r \in \Sigma^*_{obs}$ is an arbitrary sequence of observable actions. We say:*

1. *σ ends in a* normal *state if every execution sequence of G with an observable sequence σ does not have any failure events such that:*

$$\forall \mu_1 = e'_1 \ldots e'_s \in L(G) \;\; P(\mu_1) = \sigma \Rightarrow \forall i \;\; e'_i \notin f_i$$

2. *σ ends in a failure state of type f_i if every reachable sequence of events μ_2 in Σ, which can be projected to σ ends in a failure event of f_i such that:*

$$\forall \mu = e'_1 \ldots e'_s \in L(G) \;\; P(\mu_2) = \sigma \Rightarrow \forall i < s \;\; e'_i \notin f_i \text{ and } e'_s \in f_i$$

3. *σ may end in a failure state of type f_i, if some of the execution sequence of G which map to σ end in failure f_i and some have no failure of type f_i i.e.*
   $\exists \mu_1 = e'_1 \ldots e'_s, \mu_2 = e''_1 \ldots e''_q \in L(G)$ so that

$$P(\mu_1) = P(\mu_2) = \sigma \textit{ and } \forall i < s \;\; e'_i \notin f_i, e'_s \notin f_i \textit{ and } \forall i < q \; e''_i \notin f_i$$

According to the above definition, if all traces, which have the same observable events, lead to a state s_i with a normal behaviour, we can conclude that there is *no failure* in the system. However, if there are two traces τ_1 and τ_2 such that τ_1 contains a failure of F_i; whilst τ_2 does not, this means that a failure of type F_i may occur. If τ_1 and τ_2 lead to a state which has only a failure of type F_i, we can say that the failure F_i has occurred.

As explained above, a method called Diagnoser is used in order to accomplish the diagnosis of the behaviour of the system [118, 109]. In the next section, we will show how the Diagnoser is modelled and produced.

The Diagnoser model

The Diagnoser is used as a tool to detect and isolate faults in a system, which is modelled as a regular language as explained in Section 2.3.3, and captured in a finite automaton as explained in Section 2.3.2. The Diagnoser is an Automaton built from the system model in order to [118]: (i) test the diagnosability properties of the system and (ii) perform on-line monitoring of the system for the purpose of fault diagnosis.

The Diagnoser estimates of the current state of the system after the occurrence of an observable event. The Diagnoser state carries information related to failures. Such information is attached to these states in the form of *fault labels*. For example, the initial state is declared to be (x_0, N) which means that in the state x_0 the behaviour of the system is normal, while (x_1, f_1) means that the system is at state x_1 and a failure of type "1" has occurred. Consequently, as explained in Section 2.3.4, the set of failure labels $\Delta_f = \{F_1, F_2, ..., F_m\}$ are defined, where $|\prod_f| = m$ and the complete set of possible labels is

$$\Delta = \{N\} \cup 2^{\{\Delta_f\}}$$

where $\{N\}$ represents the normal behaviour, and $F_i, i \in \{1, ..., m\}$ as meaning that a failure of the type F_i has occurred.

Before presenting the Diagnoser model, we need a final piece of notation.

Notation. The states of the system G which are reached by only observable events is

$$X_{obs} = \{x_0\} \cup \{x \in X : x \text{ has an observable event into it}\} \qquad \square$$

Example 2.3.11 *In the automaton of the system described in Example 2.3.9, the states of the system G which are reached by observable events is*

$$X_{obs} = \{1, 3, 4, 5, 6, 7, 9, 10, 11, 12\}.$$

It can be seen that the state 2 and 8 are not included in this set as they are reached by unobservable events.

The Diagnoser of a system G is:

$$G_d = (Q_d, \Sigma_{obs}, \delta_d, q_0)$$

where Σ_{obs} is the finite set of observable events, q_0 is the initial state and it is defined to be (x_0, N), and Q_d is the subset of the states Q_{obs}, where $Q_{obs} = 2^{X_{obs} \times \Delta}$, composed of the states of the Diagnoser which are reached from the initial state under only observable events. Since the states of the Diagnoser is defined as a subset of states, a state q_d is of the form

$$q_d = \{(x_1, \ell_1), (x_2, \ell_2), ...(x_n, \ell_n)\}$$

where x_i is the name of the state, or $\ell_i \in \Delta$ i.e. ℓ_i is a label of the form $\ell_i = \{N\}$, and $\ell_i = \{F_{i1}, F_{i2}, ..., F_{ik}\}$. For every state in X_{obs}, we append a label which carries failure information which are used to diagnose failures.

Example 2.3.12 *In the Automaton of the system described in Example 2.3.9, the state space of Q_d can be defined as follows: $Q_d = \{q_1, q_2, \ldots, q_n\}$, where $q_1 = \{1, N\}$ which means that in state 1 the behaviour of the system is normal. At this state, the only observable event which can*

fire is α. The firing of α leads to two possible states which are the state 3 and 7, this can be computed by using the Rang function which will be explained later. The state 3 is reached by α but preceded by unobservable event σ_{f1} which represent a failure of the type f_1. This event leads to the state 2, but this state is not included in X_{obs} as it reached by unobservable event and therefore the failure is propagated from state 2 to state 3. As a result, a failure label called "F_1" is attached to the state 3. Whereas, the state 7 is reached by α and there is no failures, so a label "N" representing normal behaviour is attached to this state. Therefore, the state of the Diagnoser which can be reached after q_1 under the event α is $q_2 = \{(3, F_1), (7, N)\}$.

As explained in this example, there are two functions required to construct the Diagnoser, namely, the *Label Propagation (LP)* function and the *Range (R)* function [118]:

The Label propagation function (LP): This function modifies the failure state in the event of a failure occurrence and, as the name suggests, propagates the information about the failure in one state to a subsequent state. In layman's term, if arriving at a state x results in a failure of the type F_i, and no action is carried out to identify the failure, the following states after x will also have the failure F_i. This function is defined as follows: $LP : X_o \times \Delta \times \Sigma^* \longrightarrow \Delta$, assume $x \in X_o$, $\ell \in \Delta$, and $s \in L_o(G, x)$, the LP propagates the label ℓ over s. It is defined as follows :

$$LP(x, \ell, s) = \begin{cases} \{N\} & \rightarrow if \quad \ell = \{N\} \wedge \forall i[\Sigma_{fi} \notin s] \\ \{F_i : F_i \in \ell \vee \Sigma_{fi} \in s\} & \rightarrow otherwise \end{cases}$$

The Range function: the Range function (R) is used to compute all the reachable states after the occurrence of an event, and since the failure labels propagate from one state to another, LP is used to specify the proper label for each state. $R : Q_o \times \Sigma_o \rightarrow Q_o$ is defined as follows

$$R(q, \sigma) = \bigcup_{(x,\ell)\in q} \bigcup_{s\in L_\sigma(G,x)} \{(\delta(x, s), LP(x, \ell, s))\}$$

Figure 2.10 depicts the Diagnoser model of the system G of Figure 2.9 which is computed according to the above explanation.

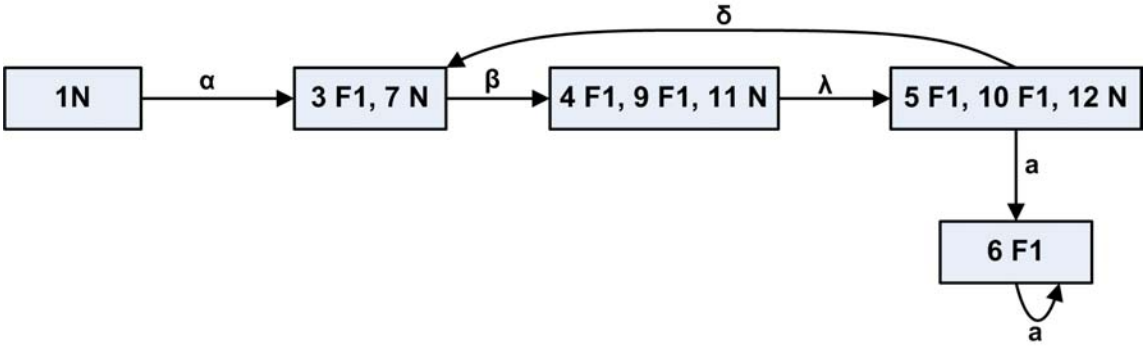

Figure 2.10: Example illustrating the construction of the Diagnoser [118]

As explained that failures in the DES context are modelled as unobservable event. Moreover, to diagnose a failure, Let say σ_{f1}, it is necessary that the system does not have a loop of unobservable events after σ_{f1} [118]. This is because the system may keep executing unobservable events after σ_{f1}, which means that no diagnosis action can be accomplished as the state of the Diagnoser never gets updated by the occurrences of unobservable events.

For further information about DES and algorithms for creating the Diagnosers Automaton, we refer the reader to [118, 119].

2.3.6 DES theory & SoA

In general, the diagnosability theory of DES have been already adopted to SoA for many purposes. For example, Yan et al. [137, 136] proposed a method to monitor Web services by tracing faults and recovering from their effects. Their method is based on using Model-Based Diagnosis (MBD) theory [62] which provides techniques to monitor static and dynamic system using partial observations. Such methods require in-depth knowledge of the system. To do so, Automaton models are extracted from the business process models. These models are used by the diagnosis method in order to reconstruct the observable trajectories of the process. When an exception is thrown, the Web service that caused the fault is deduced from the trajectory by checking the variable dependency on the trajectories. Their method is designed to deal with specific types of failures such as mismatching parameters; the occurrences of which are thrown

up as exceptions. In our approach, we deal with failures caused by undesirable scenarios, such as Right-First Time (RFT) Failure, which occurs when a business process fails to complete a task First-Time and is forced to repeat a part of the task again (i.e. when a task is executed more than once, indicating incorrect execution of the task in the first place, or the invocation of an erroneous execution). Such occurrences of failure may result in violations of Service Level Agreements (SLA), causing financial penalties or customer dissatisfaction.

Moreover, Wang et al [131] also presented approach based on applying the DES control theory in order to allow safe execution of Workflows by avoiding runtime failure. Their approach makes use of Automaton to identify forbidden states, representing in desirable execution state, to generate the control logic.

2.4 Petri nets

The Petri nets [113] is a mathematical model for modelling and analysing concurrent and distributed systems comprising synchronous and asynchronous activities. A Petri net is made of places, tokens, transitions and directed arcs. Places is the states of the system, which hold a number of tokens. A distribution of tokens on the places is known as a *marking*. A transition can occur when several conditions are satisfied. These pre-conditions are expressed as the places that are input to a transition. The occurrence of a transition affects the output places of a transition (i.e. the post-conditions).

Definition 2.4.1 *A Petri Net is a structure*

$$N = (P, T, F)$$

where P denotes the finite set of places, T denotes the finite set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions (Pre) and from transitions to places (Post), such that:

$F =$ Pre $\cup$ Post, Pre$(p, t) : P \times T \to \mathbb{N}$ and Post$(t, p) : T \times P \to \mathbb{N}$ where:

- $Pre(p,t) \in \mathbb{N}$ *gives the weight of the arc directed from place* p *to transition* t

- $Post(t,p) \in \mathbb{N}$ *gives the weight of the arc directed from transition* t *to place* p

The graphical representation of a Petri Net N represents a place as a circle and a transition as a bar (box).

Notation. For each transition t, the set of Input/Output places of t are denoted by $In(t)$ and $Out(t)$. □

A marking M of a Petri net N is captured by a $|P|$ -vector that assigns to each place p of P a non-negative number of tokens $M : P \to \mathbb{N}$. The marking is represented graphically by tokens (dots) drawn inside the circle representing the place.

Definition 2.4.2 *Consider a Petri net* N *and a marking* M*, a transition* $t \in T$ *is enabled in* M *if* $\forall p \in I(t), M(p) \geqslant Pre(p,t)$*. The set of all the enabled transitions in the marking* M *is denoted by ENABLED(M). Each enabled transition* $t \in ENABLED(M)$ *in a marking* M *can fire and produce a new marking* M' *such that:* $M' = M - Pre(p,t) + Post(t,p)$

Example 2.4.3 *Consider the Petri net graph shown in Figure 2.11. The Petri net is defined as follows:* $P = \{p_1, p_2, p_3, p_4\}$ *and* $T = \{t_1, t_2, t_3\}$*. The initial state is* $x_0 = [1,0,0,0]$*, where coordinate* i *is for the place* p_i*. The only enabled transition at this state is* t_1*. According to Definition 2.4.2, firing* t_1 *requires only one token from the place* p_1 *and we have* $M_0(p_1) = 1$ *(i.e.* $M_0(p_1) \geqslant Pre(p_1, t_1)$*). When* t_1 *fires, one token from the place* p_1 *is removed, and one token is added to each of the places* p_2 *and* p_3*. After this firing, we can obtain the new marking which is* $M_1 = [0,1,1,0]$.

In the following, we shall describe the notation of traces which represents the behaviour of a Petri net.

Definition 2.4.4 *A trace* τ *in a Petri net system* (N, M_0) *is defined as:*

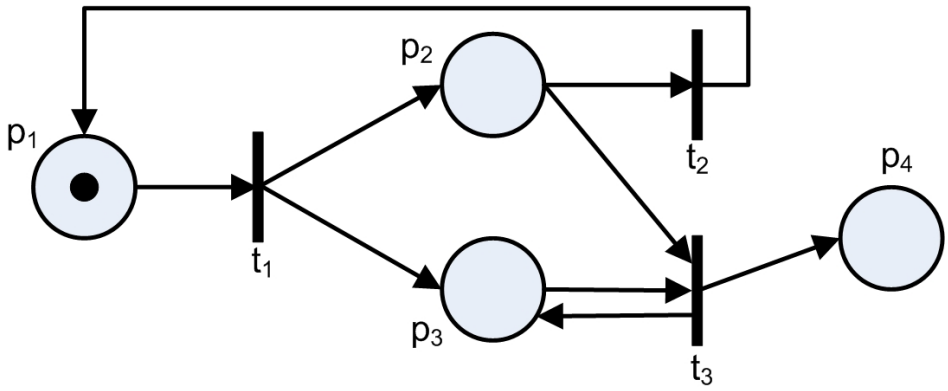

Figure 2.11: Petri Net graph of Example 2.4.3

$$\tau = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_\kappa} M_\kappa$$

where $M_i \xrightarrow{t_{i+1}} M_{i+1}$ *means that the transition* t_{i+1} *can fire and change the marking from* M_i *to* M_{i+1} *for* $i = 1, 2, \ldots, k-1$. *We can sometimes write* $M_0 \xrightarrow{\tau} M_k$ *where* $\tau = t_1 t_2 \ldots t_k$.

The Reachability of a Petri net is a very important concept for studying the dynamic properties of any system [96]. The marking of a Petri net is changed after the firing of an enabled transition. A sequence of firing results in a sequence of markings. A marking M_κ is reachable from an initial marking M_0 if there exists a trace of execution that transforms M_0 to M_κ (i.e. $M_0 \xrightarrow{\tau} M_\kappa$). Therefore, the set of all possible marking reachable from M_0 by τ is denoted by $R_N(M_0)$ called Reachable set of (N, M_0). The set of all possible firing sequence from M_0 called the language of a Petri net is denoted by $L_N(M_0)$, as it is a subset of T^*, the set of all words as in T.

Definition 2.4.5 ([113]) *Consider a Petri net N with initial Marking* M_0. *The set of all possible traces in* N *starting at the marking* M_0 *is denoted by* $L_N(M_0)$ *and the set of reachable markings is:* $R_N(M_0) = \{M | \exists \tau \in L_N(M_0)$*such that,*$M_0 \xrightarrow{\tau} M\}$

Definition 2.4.6 *(Reachability Graph). The set of reachable markings* $R_N(M_0)$ *can be represented as a graph, where the set of nodes is captured by the set of reachable markings* $R_N(M_0)$. *Each two markings are connected by an edge* $t \in T$ *such that* $M_i \xrightarrow{t} M_j$

From this definition it can be seen that if the set of reachable states is finite, it can be written as a graph.

Example 2.4.7 *Consider the Petri net of Figure 2.12 with an initial state [1, 1, 0]. All transitions, which are enabled at this state, are examined and new nodes are defined in the tree. This stage is repeated until all possible reachable states have been identified. In this example,* t_1 *is the only transition enabled at the initial marking, which leads to the next state [0, 0, 1]. In this state,* t_2 *is enabled and it leads to the next state [1, 1, 0]. It can be seen that this state already exists as the initial state, so the process is now stopped.*

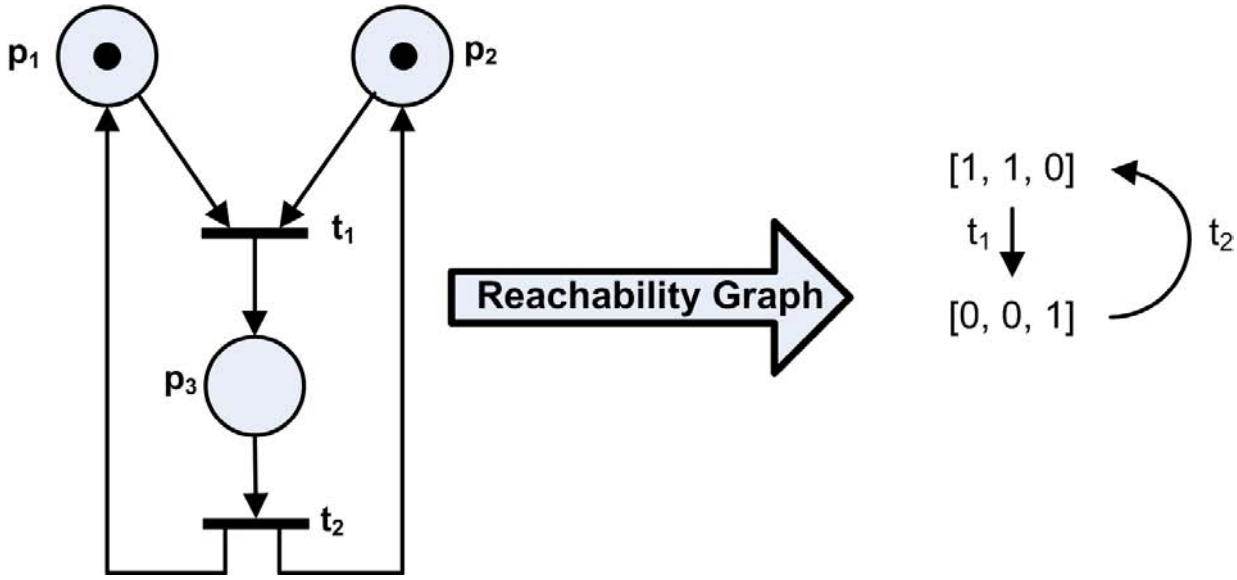

Figure 2.12: Reachability Graph of Example 2.4.7

For a more detailed description of algorithms for computing the Reachability Graph, please refer to [113].

2.4.1 Coverability Graph

The Coverability Graph is an analysis technique which may be used to solve some problems of unbounded nets, such as boundedness, safety, blocking, conservation, liveness and persistence [113]. Unbounded nets cannot be captured in a Reachability Graph as the number of states

can be infinite. Therefore, the Coverability Graph is a finite graph representing the reachable markings of Unbounded Petri net.

For a given Petri net N, we can obtain from the initial marking M_0 as many new markings as the number of enabled transitions. Then, from these new markings, we can get more markings. In this case, such a net will grow infinitely. In order to keep the graph finite, a new symbol called ω is introduced to represent unlimited capacity (i.e. $M(p_i) = \omega$ for all places $p \in P$). As a result, every Petri net can be transformed to a net with unlimited capacity and without affecting its original behaviour. Algorithm 1 can used to construct the Coverability Graph for any given Petri net. In the following example, this algorithm is applied to a simple Petri net.

Algorithm 1 The Computation of the Coverability Graph of a Petri net [96]

```
Label the initial Marking M_0 as the root and tag it "new".
while new markings exist do
  select a new marking M.
  if M is identical to a marking in the path from the root to M then
    tag M as "old" and go to another marking.
  end if
  if no transition are enabled at M then
    tag M "dead" and go to another marking.
  end if
  while there exist enabled transitions at M do
    Obtain the marking M' that results from firing t at M.
    On the path from the root to M, if there exist a marking M'' such that M'(p) ⩾ M''(p)
    for each place p and M' ≠ M'', i.e., M'' is coverable, then replace M'(p) by ω.
    Introduce M' as a node, draw an arc with label t from M to M', and tag M' "new".
  end while
end while
```

Example 2.4.8 *Consider the Petri net depicted in Figure 2.11. The initial marking M_0 of this net is $M_0 = [1, 0, 0, 0]$. Then, t_1 is enabled at M_1. Firing t_1 changes the marking from M_0 to $M_1 = [0, 1, 1, 0]$. Now there are two transitions which are enabled, namely t_2 and t_3. If t_3 fires, then the dead node $M_3 = [0, 0, 1, 1]$ is obtained, while firing t_2 changes the marking from M_0 to $M_2 = [1, 0, 1, 0]$ which covers M_0, and therefore, it is replaced with $M_2 = [1, 0, \omega, 0]$. Then, firing t_1, the symbol ω remains unchanged, and the new Marking is $M_4 = [0, 1, \omega, 0]$.*

Now, there are two transitions which are enabled, namely t_2 and t_3. Firing t_3 leads to dead node $M_5 = [0, 0, \omega, 1]$, whereas firing M_2 leads to $M_6 = [1, 0, \omega, 0]$. It can be seen that this marking already exist in the Graph as M_2, so there is no need to create a new node, but an arc labelled by t_2 is drawn from M_4 to M_2. By this stage the construction of the Coverability Graph is completed, as shown in Figure 2.13.

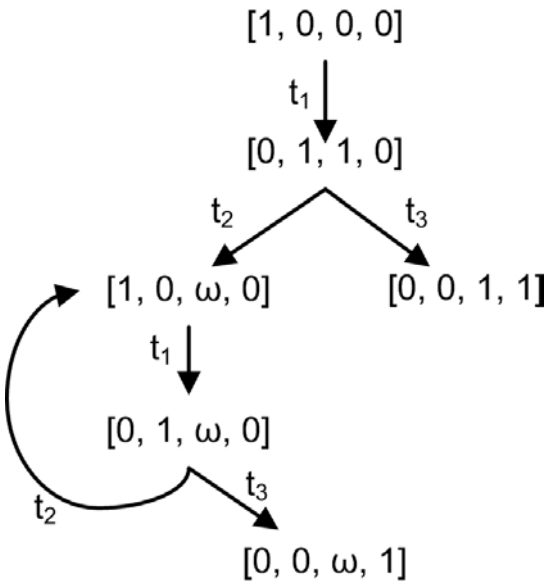

Figure 2.13: The Coverability Graph of Petri Net of Example 2.4.8

Theorem 2.4.9 ([33]) *If the Coverability Graph has no ω, then it is a Reachability Graph.*

2.5 Workflow Graph

A Workflow model [56, 44] is a collection of tasks defined to represent a business process. The Workflow model offers a flexible and appropriate environment to develop and maintain the systems in a high-level language. The main objective of Workflow models is to create high-level specifications of business processes which are independent of the Workflow management software [114].

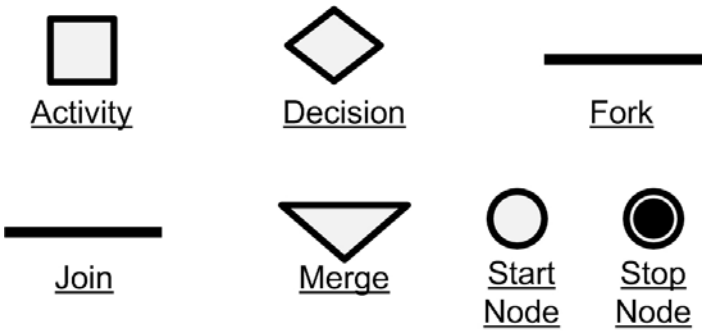

Figure 2.14: Workflow Graph Objects

There are a wide variety of approaches to Workflow modelling which have been covered in the literature [4, 127, 35, 44, 114]. The earliest methods of Workflow modelling languages are slightly ambiguous. As a result, Workflow graphs [114, 115] and Workflow nets [127] have been proposed to address such issues through involving formal methods. We find the Workflow Graph proposed by Vanhatalo et al. [128] the closest to the style adopted by major tools such as IBM WebSphere and Oracle JDeveloper. This Workflow Graph [128] extends the Workflow models introduced by Sadiq and Orlowaska [114, 115], to decompose the Workflow Graph into a net of form Single-Entry-Single-Exit (SESE). In this thesis, we make use of the improved version of the Workflow Graph [128] for modelling the business process. In this section, we recall the definition of Workflow Graphs [128].

Definition 2.5.1 *A Workflow Graph is a graph $G = (N, E)$ where N is the set of nodes and E is the set of edges. Each node $n \in N$ represents an action such as Start, Stop, Activity, Fork, Join, Decision and Merge. Each edge of the Workflow Graph connects two nodes to each other i.e. $E \subseteq N \times N$.*

Figure 2.14. shows Workflow Graph objects which are either nodes or edges. The *control flow relation* is an edge that links two nodes in a graph. The Workflow Graph has a set of nodes where each node is classified as a *task* or a *Decision/Merge* coordinator. A *Fork* node is used to accomplish concurrent executions. A *Decision* node has two or more output edges resulting in

exclusive alternative paths, which means only one alternative output is executed. A *Merge* node joins two or more inputs into one path.

Notation. Every Workflow Graph has a unique Start and Stop node. For a Workflow Graph G we shall denote them with *G.Start* and *G.Stop*. □

Notation. For each node n, the set of Input/Output edges of n are denoted by $In(n)$ and $Out(n)$. □

Fork and Decision nodes have only one input edge, while Join and Merge have only one output edge. Fork and Decision edges are generally expected to have more than one output edge, whereas Join and Merge nodes have more than one input edge.

2.5.1 Semantic of Workflow Graph

According to Vanhatalo et al. [128], the *state* of a Workflow Graph is represented by the assignment of tokens to the edges. An edge with a token indicates the action following the edge can be potentially executed. This is very similar to assignment of tokens into places in Petri net [96]. In contrast to Petri net where tokens are captured in Places, tokens in a Workflow Graph are captured in its edges. As a result, the execution of an event in a Workflow Graph moves the tokens between edges. Such movement also captures the flow of activities within a Workflow Graph.

An occurrence of an *action* n, which is modelled as a node of the Workflow Graph, denoted by $s \xrightarrow{n} s'$, which means that firing n changes the state from a state s to s'. Such changes modify the number of tokens on some edges of the Workflow Graph. Figure 2.15 represents the semantics of movement of tokens, before and after the execution of some of the nodes. For example, firing a Decision node removes one token from its input edge and adds one token to one of its output edges nondeterministically.

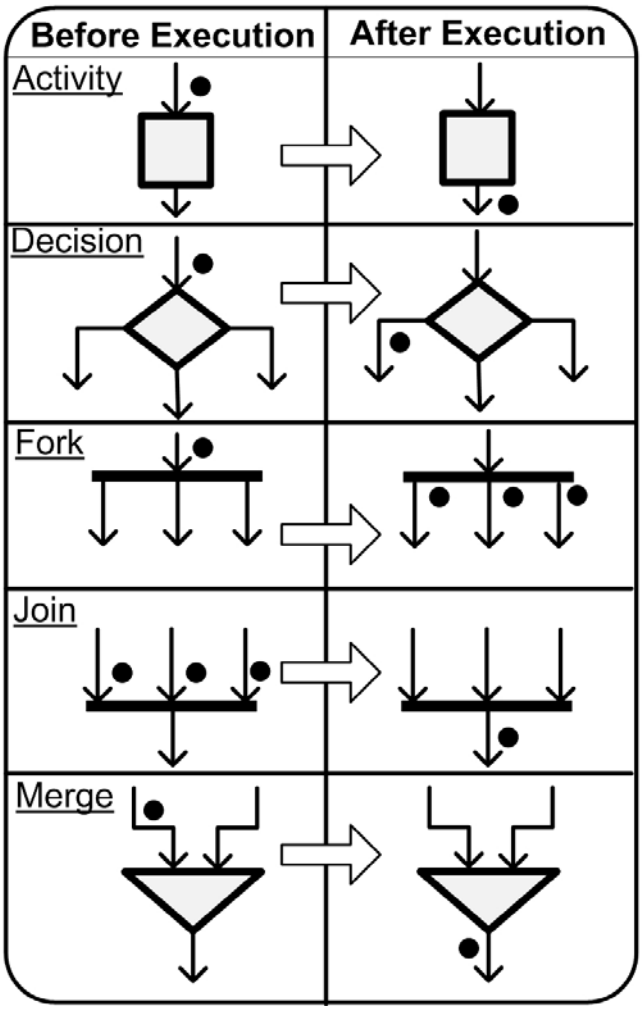

Figure 2.15: Workflow Graph Semantics

Notation. Suppose that G is a Workflow Graph. A state in G is a mapping $s : E \to \mathbb{N}$ which assigns a number of tokens to each edge $e \in E$ (i.e. $s(e) = k$ means edge e carries k tokens). An occurrence of an *action* $n \in N$ is denoted by $s \xrightarrow{n} s'$. The execution changes the state of the system from s to s'. The scenario of moving tokens between edges differs from one Node to another. □

Definition 2.5.2 *(Change of States) Assume that s, s' are two states of a Workflow Graph and $s \xrightarrow{n} s'$.*

- *if n is a Start node then:*

$$s'(e) = \begin{cases} 1 & \textit{if } e \in Out(n) \\ s(e) & \textit{otherwise} \end{cases}$$

- *if n Stop node then:*

$$s'(e) = \begin{cases} s(e) - 1 & \textit{if } e \in In(n) \\ s(e) & \textit{otherwise} \end{cases}$$

- *if n is an Activity, Fork or Join:*

$$s'(e) = \begin{cases} s(e) - 1 & \textit{if } e \in In(n) \\ s(e) + 1 & \textit{if } e \in Out(n) \\ s(e) & \textit{otherwise} \end{cases}$$

- *if n is a Decision and e' is one of the output edges of n:*

$$s'(e) = \begin{cases} s(e) - 1 & \textit{if } e \in In(n) \\ s(e) + 1 & e = e' \\ s(e) & \textit{otherwise} \end{cases}$$

  In other words, Vanhatalo et al. [128] assume that the execution of a Decision node n results in removing tokens from an input edge of n and depositing it in one of the output edges of n nondeterministically.

- *if n is a Merge and e' is one of the incoming edges of n with $s(e') > 0$:*

$$s'(e) = \begin{cases} s(e) - 1 & e = e' \\ s(e) + 1 & \textit{if } e \in Out(n) \\ s_i & \textit{otherwise} \end{cases}$$

Figure 2.16 represents a simple example of a workflow graph. It can be seen that Activities are depicted as a square, a Fork and a Join as a thin rectangle, a Decision as a diamond, and a Merge as a triangle. Start and Stop nodes are depicted as (decorated) circles. A complex example will be presented in Chapter 4. For more details on the Workflow Graph, please refer to [128].

2.6 Model Driven Architecture (MDA)

The Model Driven Architecture (MDA) [99, 51, 78] is a framework introduced by the Object Management Group (OMG) in order to promote the role of modelling in software development.

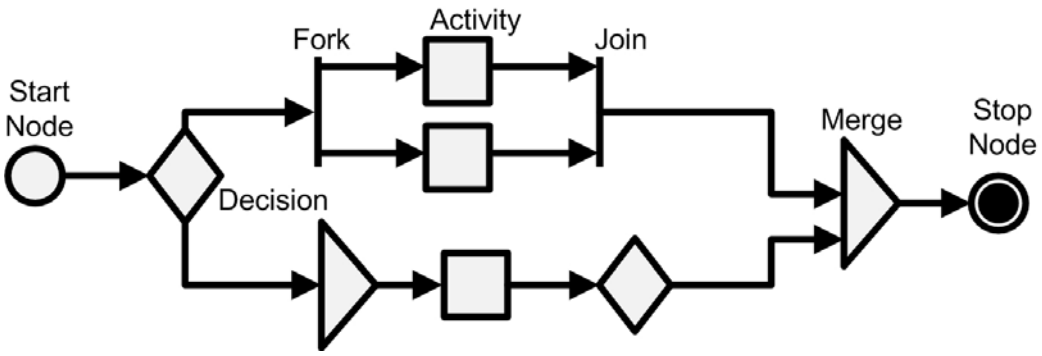

Figure 2.16: A simple workflow graph example [128]

One of the main goals of MDA is the idea of a model transformation, which maps models in a source language into models captured in a destination language. In the MDA context, a model transformation is defined by a number of transformation rules, which specify the mapping of the *meta-elements* of constructs of the *metamodel* of a *source language* into the *meta-elements* of a *destination language*. The metamodels of the source and target languages are specified using a common language called the Meta Object Facility (MOF) [93]. In general, models in the MDA are instances of metamodels. As depicted by Figure 2.17, an MDA transformation is defined from the source metamodel to the destination metamodel. Then every model, which is an instance of the corresponding metamodel, can be automatically transformed to an instance of the target metamodel. For example, to map a BPEL service to Deterministic Automaton, a model transformation that maps the metamodel of the BPEL to the metamodel of the Deterministic Automaton is required.

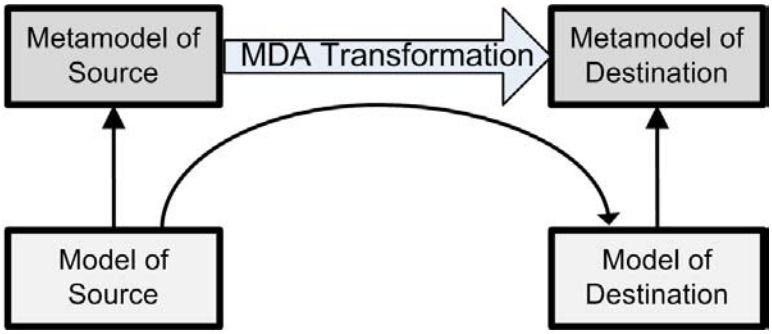

Figure 2.17: Model Driven Architecture (MDA)

Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (or QVT for short) [100] is the OMG specification which is proposed as a method to specify model transformation rules with the MOF. QVT provides a declarative and imperative language, structured into a layered architecture consisting of *Relations*, *Core* and *Operational Mappings*. The *Relations language* is a high level language that provides a textual and a graphical notation for defining the mappings, while the *Core languages* is a small language based on Essential MOF (EMOF) and OCL to support pattern matching and evaluation of conditions. The QVT *Operational Mappings language* is a high level imperative language that extends the Object Constraint Language (OCL) [101] with essential features (such as the ability to define loops) in order to write complex transformation rules [100]. In this study we shall use QVT Operational Mapping language for the specification of the transformation rules.

The QVT *Operational Mapping* language is specified as a standard way of providing imperative implementations. This language is based on using MOF as a repository for metamodels. The general syntax for the body of an Operational Mapping is depicted in Figure 2.18, where the *source* is the source of the model transformation. The *mappingFunction* is the name of the model transformation which may require some inputs which are captured by the variable *parms*. The *target* is the destination model of the transformation. The *init* part has some code which can be executed before carrying out the main body of the mapping rules. The *population* is used to populate the result of the mapping. The code included in the *end* part is executed before ending the operation. The *when* part has a boolean expression that should be true before starting the execution. The *where* part includes the conditions that have to be satisfied by the model elements involved in the mapping (i.e. It acts as a post-condition for the mapping operation).

There are many industrial and academic case tools supporting model transformations such kermeta [75], Arcstyler [15], OptimalJ [102], Xactium [135],ATLAS [18] and SiTra [6]. In this thesis, we used the Simple Transformer (SiTra) [6] transformation engine to execute the transformation rules. SiTra is a lightweight Model Transformation Framework aiming to use Java

```
mapping source::mappingFunction(parms):target
  when {...}
  where {...}
  {
        init{...}
        population{...}
        end{...}
  }
```

Figure 2.18: The general syntax for the body of a mapping operation

for both writing Model Transformations and providing a minimal environment for transformation execution. Figure 2.19 depicts an overview of SiTra. More specifically, SiTra consists of two interfaces, the *Rule* interface, which user defined mapping rules have to implement and the *Transformer* interface, which provides the skeleton of the methods that carry out the transformation. The modeller needs only to define the transformation rules by implementing the *Rule* interface, which consists of three methods: *check()*, *transform()* and *setProperties()*. If the rule is applicable for the *source* element in question, the *check()* method of the rule implementation returns true and the build() method is executed. The *build()* method generates the target model element. The *setProperties()* is used to set the attributes and links of the newly created target element. SiTra has been successfully applied to Model Transformation in various application domains [134, 29, 14, 10].

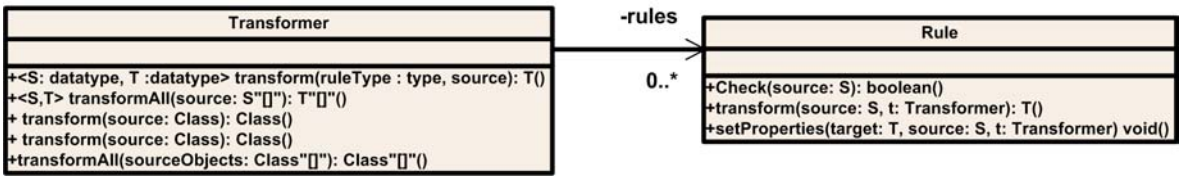

Figure 2.19: Overview of SiTra

Traceability is a feature in a Model Transformation Framework that keeps a record of which elements in the source model map to which elements in the destination. Use of tracing in MDA

has recently received considerable attention [27, 72, 121]. SiTra Model Transformation Framework [6] has been extended to support tracing for Model to Model (M2M) transformations. This extension is based on the MOF Queries Views and Transformations (QVT) specification [100], which defines how tracing can be used in Model Transformation specifications. The specification defines a number of methods that can be used to query the tracing model (i.e. a number of "resolve()" methods [100], however it does not define a specific tracing model ('*...trace classes are implicitly and automatically derived from relations*.' [100]. Therefore, a model for tracing is developed and implemented it as an extension to the SiTra Model Transformation Framework. Figure 2.20 depicts the model for tracing *M2M* transformations.

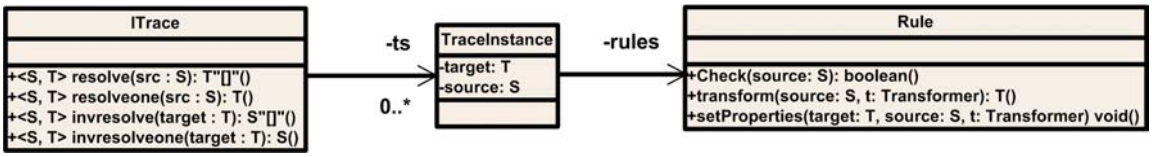

Figure 2.20: A Model of the Tracing Mechanism

In particular, the tracing consists of an interface (*ITrace*), which holds a collection of TraceInstances (ts). Each TraceInstance, represents a mapping between a source and a target model element, through a SiTra rule. An implementation of the *ITrace* interface, provides a number of methods to query the ts collection. More specifically the *resolve* method, queries the ts collection, and returns all target instances that have been created during the transformation, from the *src* instance. Likewise, *resolveone* should return only the first instance of the target element that has been created during the transformation from the *src* instance. The method names preceded with "inv" (i.e. *invresolve*), perform the inverse (i.e. return the source elements that have been mapped to the target element passed as a parameter). For further details on SiTra please refer to [6].

CHAPTER 3

ANNOTATING BPEL MODELS & MODELLING FAILURE

As explained in Section 1.3, this work proposes an approach based on adopting the diagnosability theory of Discrete Event System (DES) to SoA in order to automate the creation of the Diagnoser. Our approach has been achieved as follows: firstly, BPEL models are annotated with new information to deal with Observability. This is considered a requisite task before applying the theory of DES. Secondly, the Workflow Graph [128] has then been adopted and extended as a language for specifying models of business processes, as will be explained in Chapter 4. Then, the Diagnosability theory of DES has been adopted to the Extended Workflow Graph in order to produce the Diagnoser, as will be described in Chapter 5. Fourthly, various methods are proposed to implement and integrate the produced Diagnoser into the system, as will be discussed in Chapter 6. Finally, the presented approach is implemented as a Plugin for Oracle JDeveloper. Our implementation will be discussed in Chapter 7.

In this chapter, we discuss the first step which is the annotations required to adopt the DES theory in SoA. This includes describing how to annotate a BPEL activity with new information to allow specifying observable and unobservable events according to the definition of the Diagnosability of DES, as will be explained in Section 3.2.1. As explained in Section 2.3.4,

the failures are modelled as a part of the business process with unobservable events. Therefore, Section 3.2.2 describes how to annotate a BPEL activity to represent a failure. These annotations have led to the extension of the current BPEL metamodel, which will be used in our approach when we automate the creation and integration of the Diagnoser. This extension will be discussed in Section 3.3.

Before explaining our approach, a simple running example will be introduced, for demonstration purposes.

3.1 A Running example: The e-shopping system

In this section, a modified version of the example given by Guillou et al. [85, 86] will be provided in order to demonstrate our approach. This example is based on a typical on-line, e-shopping system consisting of three main services namely, Shop, Supplier and Warehouse.

As depicted in Figure 3.1, the customer accesses the website of the shop in order to search the available items. They add items to the shopping cart which is subsequently passed to the Supplier service. For each item in the list, the Supplier service sends a request to the warehouse to check if the item is currently available. If it is available, the Warehouse service sends an acknowledgement to the Supplier to complete the processing of the order. The Supplier service then sends back the list of available items to the Shop service; and finally the customer is asked to confirm their order.

Figure 3.2 depicts the BPEL service of the Shop service. It can be seen that the Shop service receives the placed order which is then stored in its database. The Shop service will then send the list of items to the Supplier service in order to check availability. After receiving the list of available items from the Supplier, the total amount that should be paid is calculated and a summary of the order is sent to the customer. Consequently, the customer has the option to either confirm or cancel their order.

Figure 3.3 depicts the BPEL of the Supplier service which receives the list of items from

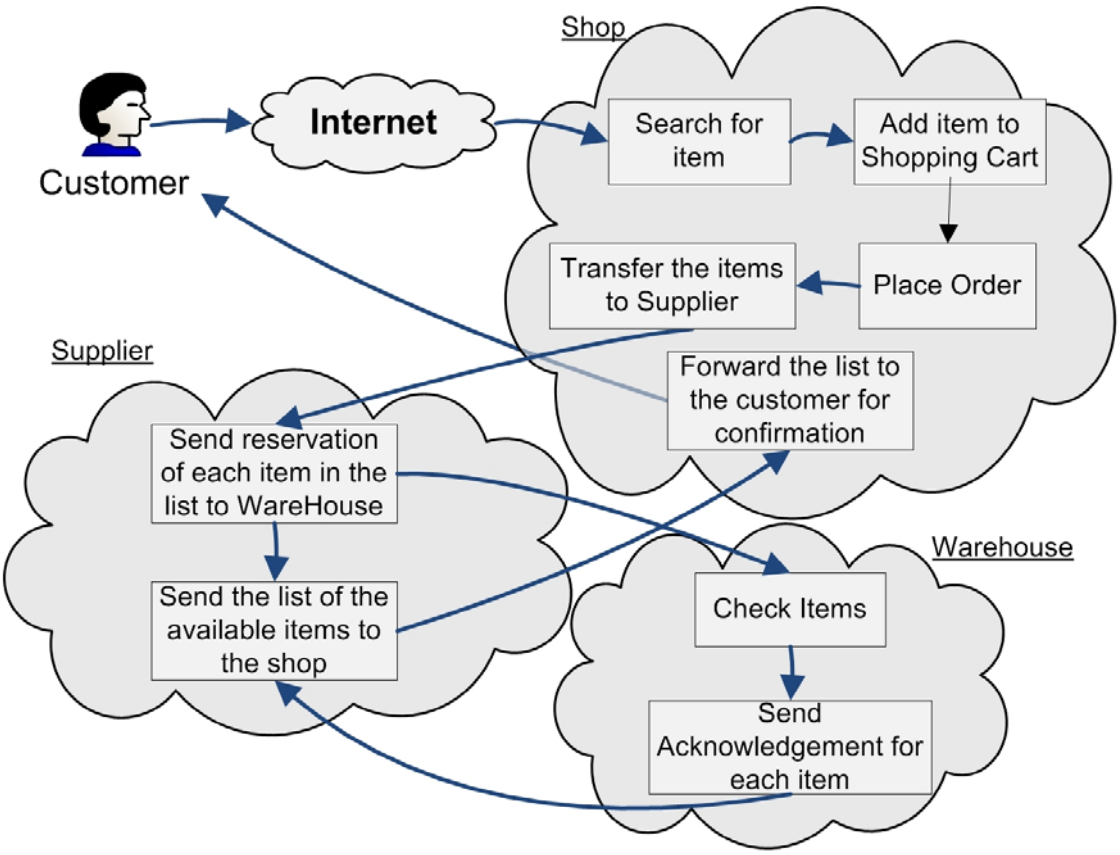

Figure 3.1: E-shopping scenario

the Shop service. A request for each item in the list is sent to the Warehouse service. If the item is available at the warehouse, the list of items is updated. The final updated list is returned back to the Shop service after iteration of processed items.

Figure 3.4 depicts the BPEL of the Warehouse service which receives the request sent by the Supplier for each item. If the item is available, it is reserved and subsequently the list of items in the stock is updated. If not, the Supplier is informed that the item is not available.

3.2 Annotating BPEL

In order to adopt DES techniques in BPEL, we annotate BPEL representations with new information which allows the client to identify which activities are observable or unobservable, and

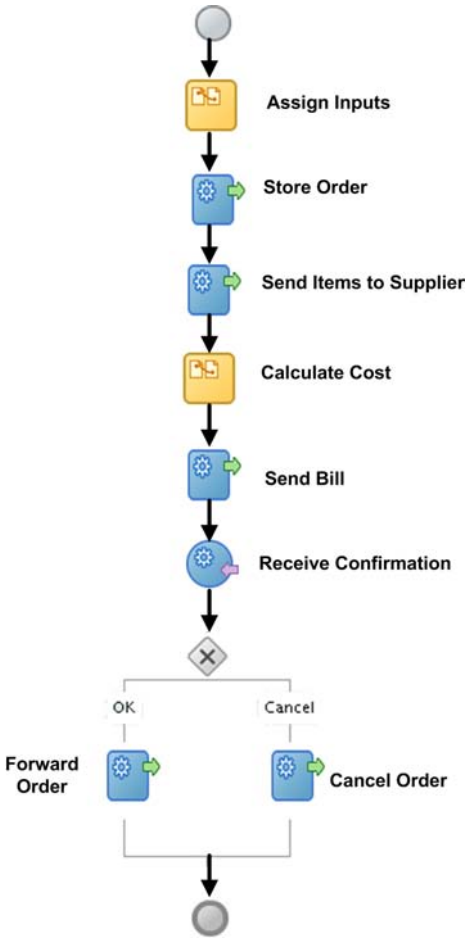

Figure 3.2: The Shop BPEL service

which events represent failure actions. Such information is not included in the current BPEL model; a common practice is to annotate the BPEL file to include such information. To achieve this, two main annotations are conducted: the first is to annotate activities to have information related to Observability, while the second is to annotate activities to represent failures. Next, we will explain these annotations in detail and illustrate them with the help of an example.

3.2.1 Annotating Activities

In the context of DES, *Events* are actions which change the status of the system from one state to another. Some of these events are considered observable event, i.e. their occurrences can be

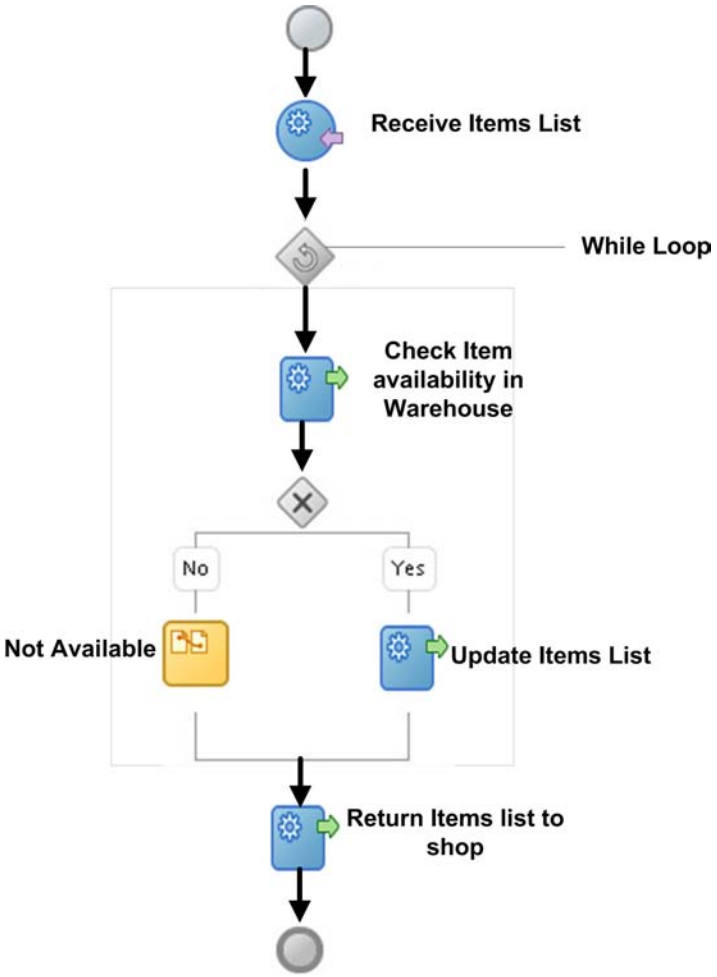

Figure 3.3: The BPEL service for the Supplier services

seen by external components, while others are considered as unobservable i.e. their occurrences remain hidden from other components, as explained in Section 2.3.4.

BPEL activities can be described as being similar to DES actions, but require minor annotations. Such annotations include specifying observable and unobservable events. In our approach, this is accomplished by adding a new attribute, which is called *xml:IsObservable*, to the construct of each BPEL activity. Consequently, a BPEL activity can be defined as an observable event by assigning "yes" to the value of the *xml:IsObservable* attribute, or "no" if it is unobservable. Figure 3.5 shows an example of how to annotate the construct of an *Invoke*

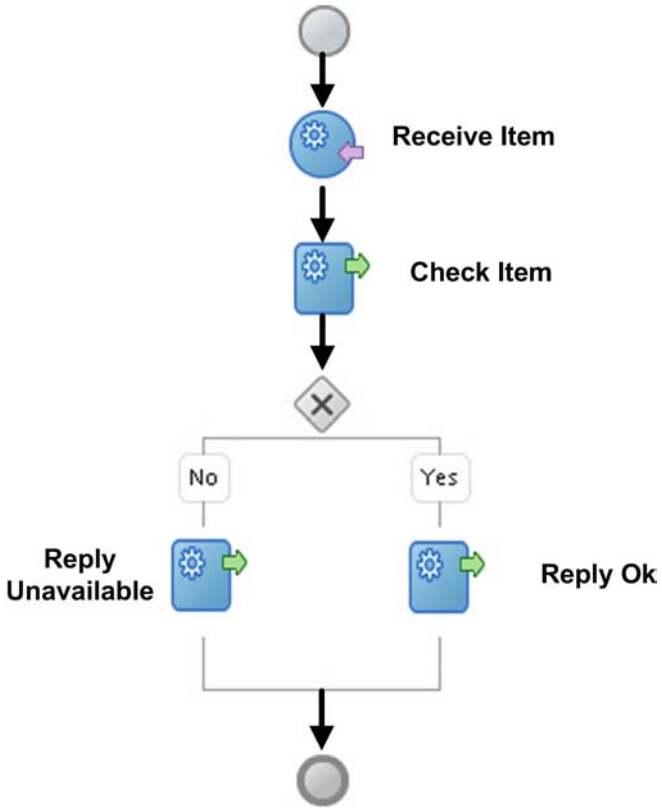

Figure 3.4: The BPEL of the Warehouse Service

activity, called *SendItemsToSupplier* of the BPEL model of the Shop service depicted in Figure 3.2, to include such annotations.

3.2.2 Failures

SoA-based systems can fail because of the failure of the underlying services or hardware resources. Another source of failure is due to or revealed by data such as mismatching parameters; the occurrences of which are thrown up as exceptions or QoS deviations. Such failures have been well studied and treated by adopting well-established techniques, such as Model-based Diagnosis theory [62, 137, 136, 16], WS-Policy [19, 74, 123, 20], interception of SOAP messages [95] and common techniques of Quality of Service (QoS) [38]. Such failures fall out of the scope of this research.

```
<invoke name="CheckCustomerAccount"
partnerLink="CustomerProcess"
portType="ns1:CustomerProcess" operation="CheckCustomerAccount"
xml:IsObservable="yes"/>
```

Figure 3.5: Observability Annotations of BPEL Activity

Another important source of failure, which is our target in this thesis, is through the execution of an undesirable sequence of actions. Such cases are often modelled as a part of the business process, but represent cases where the execution of events produces an undesirable result. For example, Right-First Time (RFT) Failure is a typical failure in telecommunication companies, which occurs when a business process fails to complete a task First-Time and is forced to repeat a part of the task again (i.e. when a task is executed more than once, indicating incorrect execution of the task in the first place, or the invocation of an erroneous execution). Such occurrences of failure may result in violations of Service Level Agreements (SLA), causing financial penalties or customer dissatisfaction. As mentioned above, in this thesis we are dealing with such types of failures that can be modelled as a part of of the business process. We can imagine an activity in the system that represents such failures. In line with DES, if such an activity is observable, its detection would be trivial. Therefore, this activity should be considered unobservable and constitute occurrence of a failure in the same time.

Inspired by DES, a BPEL activity which is considered as a failure action, is marked by annotation. This annotation is involved in a similar way as described in Section 3.2.1. To achieve this, two further attributes, called *xml:failureEvent* and *xml:typefailure*, are added to the construct of the BPEL activity. The *xml:failureEvent* attribute is used to declare that the activity is indeed a failure. The *xml:typefailure* attribute is used to represent the type of the failure. Figure 3.6 shows an example of how to annotate the construct of an Invoke activity,

called *GES_RFT*, which is assumed to be a failure event. It can be seen that the *xml:failureEvent* attribute is assigned to "yes" to indicate that this activity is a failure; the *xml:typefailure* attribute is assigned to "1" to indicate the type of the failure.

```
<invoke name="GES_RFT" partnerLink="GeneralEvaluationService"
operation="process"
xml:IsObservable="no"
xml:failureEvent="yes" xml:typefailure="1"/>
```

Figure 3.6: Failure Annotations for BPEL Activity

In this thesis we do not focus on how to model failures as it is a topic of its own. This remains for our future research, as we will try to find a method to automate the modelling of failures and integrate this method with our approach, as will be discussed in our future research in Section 9.2. In this thesis, failures are considered to be manually identified by the client as unobservable events or as undesirable traces. Other type of failure such as violation of constraints related to value of some attributes are not directly considered. However, we can imagine modifying the model to include activities, where their occurrences represent violation of a constraint.

3.3 Annotated BPEL Metamodel Specification

As explained in Section 3.2.1 and 3.2.2, adopting the diagnosability theory of DES to SoA requires few annotations. Such annotations are not included in the current version of the BPEL metamodel [30]. Therefore, we have extended the BPEL metamodel [30] to include these annotations. This is achieved by adding the presented annotations as attributes to the *Activity meta-elements* of the BPEL metamodel, as shown in Figure 3.7. These additional *attributes* are marked with (*) in Figure 3.7. It can be seen that *Invoke*, *Reply*, *Receive* and *Assign* activities inherit new attributes, namely *IsObservable*, *IsFailure* and *typeFailure* from the Activity meta-elements.

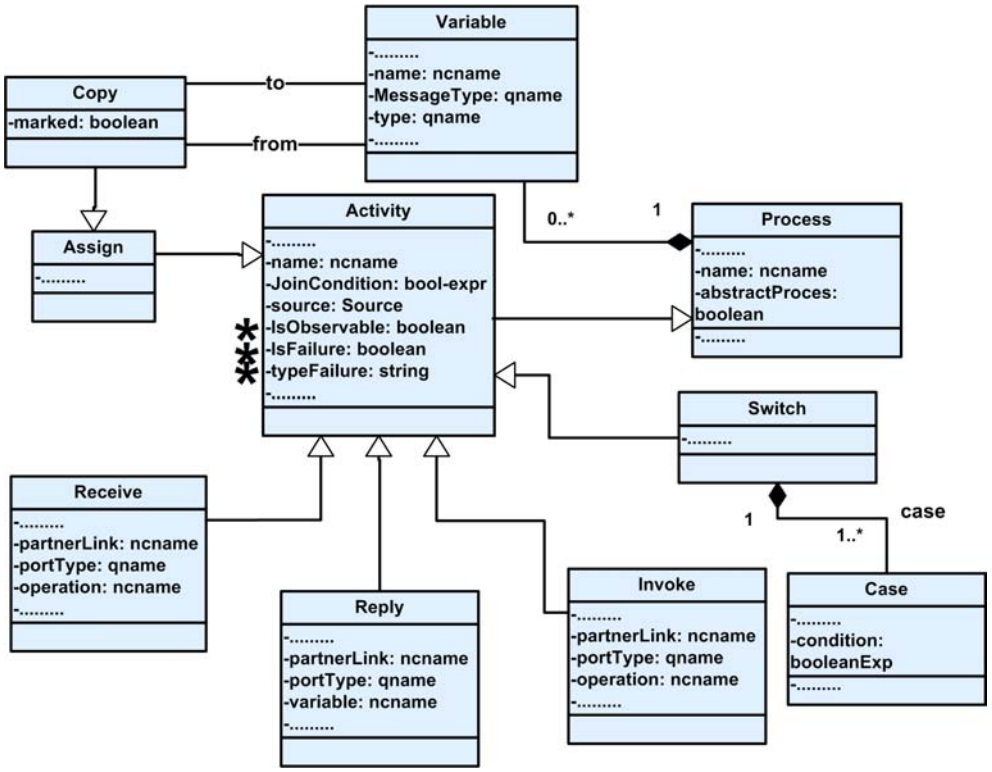

Figure 3.7: A fragment of a BPEL metamodel with added elements marked by (*)

These annotation will be used when we extract equivalent Workflow Graphs for the BPEL models in order to compute the Diagnoser, as will be explained in the following chapters.

One issue which falls outside of the scope of this thesis was a systematic way to annotate BPEL. Such annotations are manually inserted in the current version of our tool. However, we can imagine an edition which automates the process of annotating activities. This can be achieved by developing a GUI which helps the user to tick the events and mark them either observable or unobservable action.

CHAPTER 4

A MODELLING APPROACH TO THE BUSINESS PROCESSES

As explained in Section 1.3, adopting the Diagnosability theory of DES to SoA requires coming up with an appropriate modelling language framework. This has been achieved by the adoption of the conventional Workflow Graph, which closely follows the BPEL standard. This results in high level models which are similar to models used in DES such as Automaton and Petri net. Section 4.1 presents an overview of our approach. The idea behind adopting the conventional Workflow Graph is explained in Section 4.2. Our adoption has included extending the Workflow Graph to incorporate certain unsupported BPEL constructs, such as the While loop. Indeed, it is possible to express repetitive behaviours in the Workflow Graph by linking the flow back to an early node. However, this type of cycle, known as *unstructured loops*, are not supported and are therefore not recommended by the BPEL tool vendors such as Oracle JDeveloper and WebSphere. Section 4.3 describes the idea behind of *unstructured loops*. Section 4.4 presents the Extended Workflow Graph, which we have proposed to avoid such *unstructured loops*. The notation of state and the semantics of the Extended Workflow Graph are presented in sections 4.4.1 and 4.4.2 respectively.

4.1 Overview

The formulation of an appropriate modelling language framework is considered a prerequisite for utilising DES techniques. Workflow models are now widely used for the specification of business processes [4, 127, 35, 114]. Van der Aalst et al. [4, 127] and their analysis of the systems, presented a Workflow modelling language in which models were constructed from blocks of Petri net representing common workflow constructs. The blocks of Petri net were assembled together to create new representations of the overall business process. These new models were used in conducting analysis in order to identify, amongst other things, the existence of deadlocks. Petri nets, despite being a powerful representation, have not been adapted by the SoA community; who have shown greater enthusiasm for modelling via BPEL [73] and BPMN [7].

Such languages have also been supported by tool vendors. The aim has not only been to deal with all the complex and elaborate constructs supported by such tools; but to capture an essential core of the high-level interactions such as sequential, parallel and decision behaviour. Therefore, we have adopted the Workflow Graph suggested by [128], which closely follows BPEL standards, as a language for specifying models of business processes. The presented formalism directly maps models used within industry into the business process.

4.2 Adapting the Conventional Workflow Graph

As explained in Section 2.5, the Workflow Graph is considered a rich modelling language which includes necessary constructs, such as Fork, Join, Merge and Decision, commonly used in the modelling of business processes. The modelling language suggested by Vanhatalo et al. [128], based on Petri net, has been found to be the closest in style to those adopted by the major tools such as IBM WebSphere and Oracle JDeveloper. This model supports most of the complex constructs of BPEL such as sequential, parallel and decision behaviours, as described in Section

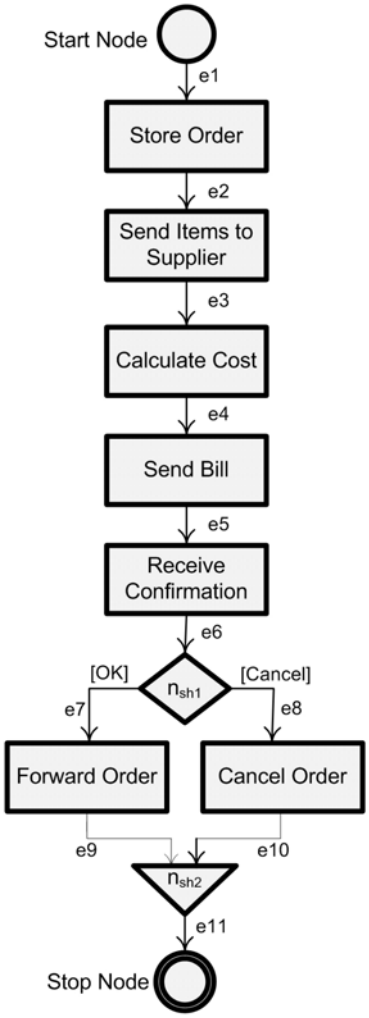

Figure 4.1: The Workflow Graph for the Shop service

2.5. For example, the BPEL models of the running example described in Section 3.1, which are the Shop, Supplier and Warehouse, can be modelled as Workflow Graphs as depicted in Figures 4.1,4.2, and 4.3 respectively. Moreover, Figure 4.4 depicts another example of how to represent a flow activity in a Workflow Graph with the help of Fork and Join nodes.

A repetitive scenario can be expressed in the conventional Workflow Graph by creating loops. This is done by connecting a node to an earlier one. However, creating loops by making cycles in the Workflow Graph may lead to *unstructured cycles* which are not supported by BPEL tool vendors.

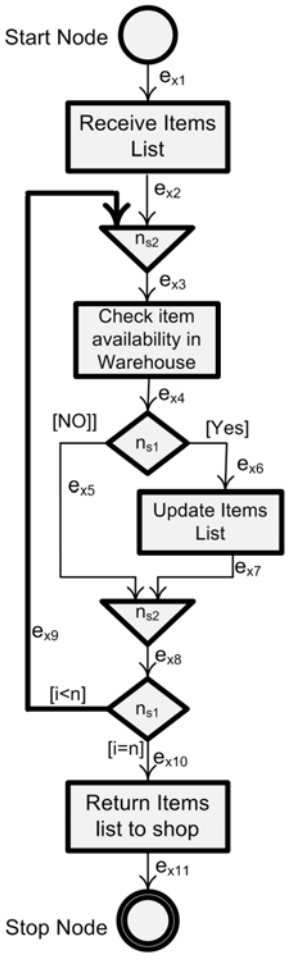

Figure 4.2: The Workflow Graph for the Supplier services

In the following sections, the *unstructured cycles* and how we have extended the conventional Workflow Graph of [128] to avoid such loops, will be described.

4.3 Unstructured loops

In the Workflow Graph of Vanhatalo et al. [128], a repetitive behaviour can be represented by a cycle in the graph. This is created by the combination of a Merge and Decision to link a node to an earlier one on the path from the Start node. For example, the iterative scenario related to the checking of availability of each item in the list of a customer's order could be captured as

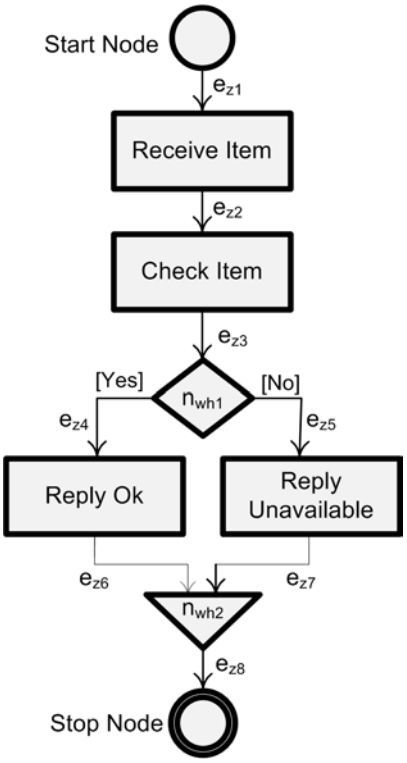

Figure 4.3: The Workflow Graph of the Warehouse Service

depicted in Figure 4.2. This style denotes repetitive behaviour, known as an *unstructured loop*, similar to the use of the *goto* command in programming languages which permits loops with multiple entry and exit.

In programming languages, the use of the *goto* command has been discouraged as it can render the system unreliable, unreadable and hard to debug [39, 43]. The creation of Workflow Graph models, which involve *unstructured loops*, may result in the elimination of parallel behaviour thus resulting in inefficient code [13]. In addition, linking flows back in the model to a previous activity may cause problems with both simulation and exporting of the Workflow Graph to BPEL [50, 79].

There are various algorithms available which allow the elimination of *unstructured loops* and replace them with equivalent loops with a single input and output node [80, 13]. Such eliminations do not however reduce the expressive power of the Workflow Graph. Leading tool

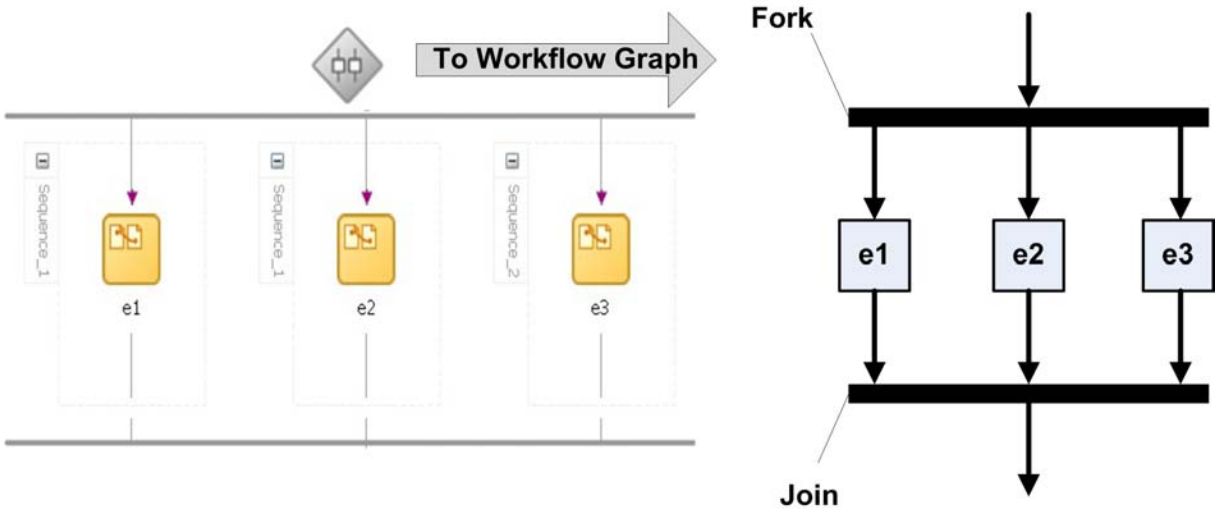

Figure 4.4: Representation of Flow activity in the Workflow Graph

vendors make use of such algorithms to eliminate the creation of *unstructured loops* in a model. For example, it is not possible to produce a Workflow Graph similar to Figure 4.2 in Oracle JDeveloper. Instead, such tools adopt a simple algorithm [13] for normalising the control-flow of the process to produce a hierarchical style similar to "While loop" in conventional programming languages [50, 79, 73].

A While loop is designed to repeat a set of tasks as long as the loop *condition* is deemed valid. It is also worth mentioning that BPEL does not support For-Loops or Do-While Loops because they would prevent the process from being exported [50, 79]. As a result, a repetitive behaviour such as the Supplier behaviour captured in Figure 4.2 must be converted to a While node as depicted in Figure 4.5; where its internal repetitive behaviour is captured in a separate Workflow Graph (*B : While Block*).

However, the formalism presented by Vanhatalo et al. [128], although closely follows the standard, does not support such While loop constructs. In the following section, we shall explain how we extended this formalism to include such actions.

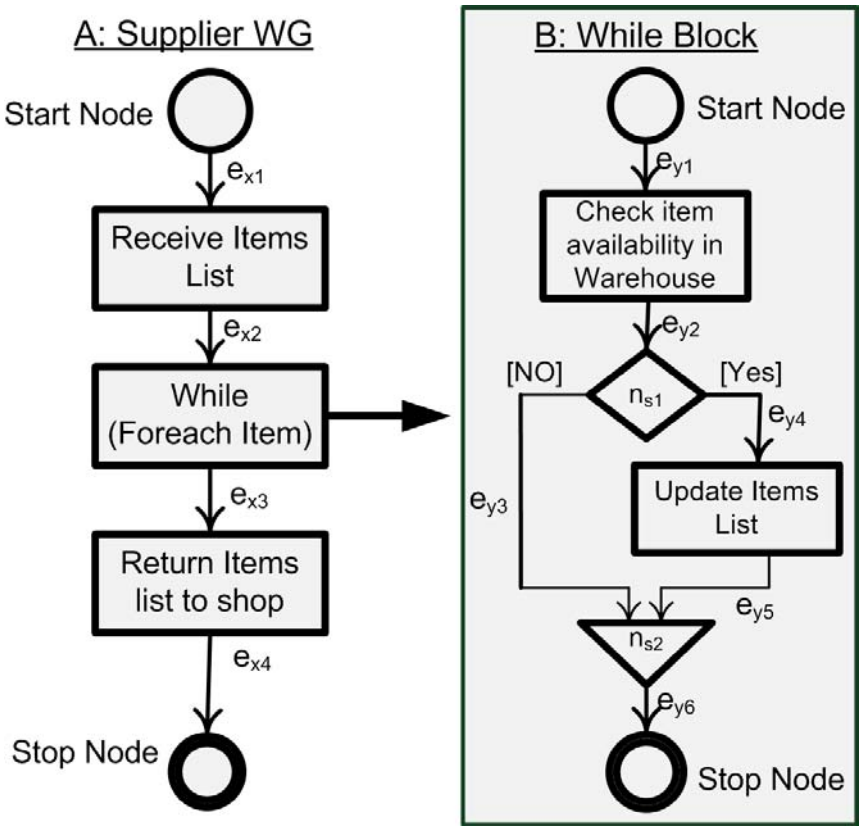

Figure 4.5: A Workflow Graph of the Supplier service with While node

4.4 The Extended Workflow Graph model

In this section, we shall extend the formalism presented by Vanhatalo et al. [128] to support While loop of BPEL. The Invocation node which represents the Invoke activity of BPEL, used to perform remote invocations of other services, will also be introduced [73]. The execution of an Invocation node can be either a *one-way operation* (i.e. it has only an input message and does not expect a result to be returned from the remote service); or a *two-way operation* which has an input message and should return a *result* synchronously [73].

The extension of the Workflow Graph model of [128] has been achieved in two steps. Firstly, an enhancement of the conventional Workflow Graph which we have called A *Workflow Graph with Invocation and While nodes (WFGIW)* is introduced. The WFGIW is an extension of the

Workflow Graph to include the constructs of the Invocation and the While node. The WFGIW is used at the nodes of the tree that represent, for example, the relation between a While loop and its repetitive behaviour. Similarly, there is an internal behaviour which itself can be a WFGIW for each Invocation node. As a result, modelling business processes in our model are in the form of a tree with a WFGIW on each node. Secondly, in definition 4.4.3 the tree representation has been elevated to introduce the notation of the *Extended Workflow Graph*.

Definition 4.4.1 *A Workflow Graph with Invocation and While nodes (WFGIW) is a graph* $G = (N, E)$ *such that:*

(I) N *is the set of nodes representing one of the following: Start node, Stop node, Activity, Fork, Join, Decision, Merge, Invocation and While.*

(II) The set of all Invocation nodes of G *is denoted by* $\mathscr{I}(G)$.

(III) The set of all While nodes of G *is denoted by* $\mathscr{W}(G)$.

(IV) E *is the set of edges such that* $E \subseteq N \times N$ *where each edge* $e \in E$ *connects two nodes with each other.*

(V) Invocation and While nodes have a single input and a single output edge which are not identical such that:

$$\forall n \in \mathscr{I}(G) \bigcup \mathscr{W}(G) \ |In(n)| = |Out(n)| = 1 \text{ and } In(n) \bigcap Out(n) = \phi$$

Example 4.4.2 *The repetitive behaviour of the Workflow Graph of Figure 4.2 can be expressed with the help of the Workflow Graph with Invocation and While nodes. Figure 4.5 depicts the Supplier WFGIW in (A: Supplier WG). The internal behaviour associated to the execution of the While node is captured in another WFGIW as depicted in (B: While Block). Such an internal*

behaviour for the While node can be executed as long as the condition attached to the While node is true[1].

Definition 4.4.3 *An Extended Workflow Graph (EWFG) is a tree of form* $T = (\mathscr{V}, \Sigma)$ *where* $\mathscr{V} = \{G_1, \ldots, G_n\}$ *is the set of Workflow Graphs with Invocation and While nodes (WFGIW).* Σ *is the set of the edges of the Extended Workflow Graph. Each edge of the tree maps two Workflow Graphs together* $e = (n, G_j) \in \beta$ *where* β *is a function that maps Invocation or While nodes to their corresponding internal behaviour such that:*

$$\beta : \bigcup_{i=1}^{n} \mathscr{I}(G_i) \cup \mathscr{W}(G_i) \longrightarrow \mathscr{V}$$

For each $G_i \in \mathscr{V}$*, except the root node,* $\beta^{-1}(G_i)$ *is the unique Invocation or While node associated to* G_i*.*

The following properties can be inferred from the above definition:

1. There are no Invocations or While nodes in the final Workflow Graph of an EWFG because T is a finite tree.

2. If there is more than one Invocation or While node in a Workflow Graph, there are more than one edge out of that Workflow Graph. To be precise, for each Workflow Graph G_i the number of Invocation and While nodes is exactly the same as the number of edges starting at G_i. This ensures the assignment of a unique WFGIW to each Invocation or While node as their internal behaviour.

3. Invocation and While nodes cannot call a WFGIW which is their father node, because T is a tree.

[1]Modelling of such condition requires modelling of data. The focus of this research is on diagnosing the failure related to the flow of actions. Modelling of data remains a topic for future research. In the absence of such a condition, we will assume that an internal behaviour can occur any arbitrary number of times.

Clearly, the conventional Workflow Graph is a special case of the Workflow Graph with Invocation and While nodes (WFGIW). Occasionally, when there is no chance of ambiguity, we will intend to abuse the notation and refer to a WFGIW as simply a "Workflow Graph". However, the phrase Extended Workflow Graph will always be used for the tree which has WFGIWs as its nodes.

According to our extension, the state of the conventional Workflow Graph should be extended. In order to achieve this, the State notation has been enriched with new information to capture the exact state of the Extended Workflow Graph. In the following section, we shall introduce the notation of *State* associated with the Extended Workflow Graph.

4.4.1 States of the Extended Workflow Graph

Vanhatalo et al. [128] defined the state of a Workflow Graph G as a function $s : E \longrightarrow \mathbb{N}$, where E is the set of edges in G. We have enhanced this definition for the Extended Workflow Graph which has a multiple set of edges $E_1, \ldots, E_n$. We therefore can write $E = E_1 \cup \cdots \cup E_n$; and have defined a function $s^E : E \longrightarrow \mathbb{N}$ for capturing the number of tokens on each of the edges of the Extended Workflow Graph $T = (\mathscr{V}, \Sigma)$ where $\mathscr{V} = \{G_1, \ldots, G_n\}$ and $E_1, \ldots, E_n$ are edges of $G_1, \ldots, G_n$. However, this would be inadequate for capturing the States of the Extended Workflow Graph. Further explanation will be provided therefore with the help of the Workflow Graph in Figure 4.5. If one of the edges in block B is marked with a token, we can infer that the internal Workflow Graph related to the While loop is presently executing. Therefore, we need a mechanism to capture this information in the Workflow Graph as represented in block A. To achieve this, we have proposed two methods to assign a token to each While and Invocation node, which its internal Workflow Graph is presently executing. As a result, a State of the Extended Workflow Graph consists of a trio of functions (s^E, s^I, s^w) where s^E is the extended state, as explained above; s^I and s^w are functions to capture the number of tokens of Invocation and While nodes respectively. This can therefore be formalised in the following manner:

Definition 4.4.4 *Suppose $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph EWFG where $\mathscr{V} = \{G_1, \ldots, G_n\}$. For each i, E_i, $\mathscr{I}(G_i)$ and $\mathscr{W}(G_i)$ represent the set of edges, Invocation and While nodes of G_i. Each state s of T is a trio of functions $s = (s^E, s^I, s^w)$ such that:*

(I) $s^E : E_1 \cup \cdots \cup E_n \longrightarrow \mathbb{N}$, *where* $E = E_1 \cup \cdots \cup E_n$ *and* $s^E(e) = n$ *means that there are* n *tokens on the edge* e

(II) $s^I : \bigcup_{i=1}^{n} \mathscr{I}(G_i) \longrightarrow \{0, 1\}$, *where* $s^I(n) = 1$ *if and only if the Workflow Graph representing the internal behaviour of the Invocation node* n *is executing.*

(III) $s^w : \bigcup_{i=1}^{n} \mathscr{W}(G_i) \longrightarrow \{0, 1\}$, *where* $s^w(n) = 1$ *if and only if the Workflow Graph representing the internal behaviour of the While node* n *is executing.*

Notation. We will write each trio of functions $s = (s^E, s^I, s^w)$ as the concatenation of the three parts involving coordinates which are mapped to edges $E_1 \cup \cdots \cup E_n$, Invocation nodes $\bigcup_{i=1}^{n} \mathscr{I}(G_i)$ and While nodes $\bigcup_{i=1}^{n} \mathscr{W}(G_i)$. □

Example 4.4.5 *The simple example explained in Section 3.1 will be used to illustrate the State of the Extended Workflow Graph with the help of the initial state. This Extended Workflow Graph is a tree $T = (\mathscr{V}, \Sigma)$, where $\mathscr{V} = \{G_1, G_2, G_3, G_4\}$ such that: G_1,G_2,G_3 and G_4 are Workflow Graphs respectively corresponding to Shop, Supplier; the Workflow Graph associated to the While node of the Supplier and Warehouse. Consequently, the structure of the state is divided into four parts, each of which captures the movement of the tokens in the Workflow Graph associated to that part. These parts can be explained in following manner:*

Shop Part:

$$\underbrace{\overset{e_1}{1}\overset{e_2}{0}\overset{e_3}{0}\overset{e_4}{0}\overset{e_5}{0}\overset{e_6}{0}\overset{e_7}{0}\overset{e_8}{0}\overset{e_9}{0}\overset{e_{10}}{0}\overset{e_{11}}{0}\overset{s_1^I}{0}}_{\textit{Shop}, |s|=12}$$

The first part captures the behaviour of the Shop Workflow Graph depicted in Figure 4.1. This part has 12 *coordinates whereby* 11 *of these are the total number of the edges of the Shop*

Workflow Graph. There is one additional coordinate for the Invocation node which is used to represent the invocation of the Supplier Workflow Graph.

Supplier Part:

$$\underbrace{\overset{e_1 e_2 e_3 e_4 s_1^W}{0\,0\,0\,0\,0}}_{\textit{Supplier},|s|=5}$$

The second part captures the behaviour of the Supplier Workflow Graph depicted in Figure 4.5A. This part has 5 *coordinates whereby* 4 *of these are the total number of the edges of the Supplier Workflow Graph. There is one additional coordinate for the While node which is used to check the availability of the list of items with the Warehouse.*

While Part:

$$\underbrace{\overset{e_1 e_2 e_3 e_4 e_5 e_6 s_1^I}{0\,0\,0\,0\,0\,0\,0}}_{\textit{While node},|s|=7}$$

The third part represents the Workflow Graph depicted in Figure 4.5B which captures the internal behaviour of the While node of the Supplier Workflow Graph. This part has 7 *coordinates whereby* 6 *of these are the total number of the edges of the Workflow Graph. There is one additional coordinate for the Invocation node which is used to invoke the Warehouse Workflow Graph.*

Warehouse Part:

$$\underbrace{\overset{e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8}{0\,0\,0\,0\,0\,0\,0\,0}}_{\textit{Warehouse},|s|=8}$$

The last part represents the Warehouse Workflow Graph as depicted in Figure 4.3. This part has 8 *coordinates, which are the total number of the edges of the Warehouse Workflow Graph.*

As a result, the state of the Extended Workflow Graph of the given example is expressed as a combination of these parts. Figure 4.6 depicts the initial state of the system which has only one token in the output edge of the Start node of the Shop Workflow Graph. This is indicated by assigning "1" to the first coordinate of the Shop Workflow Graph.

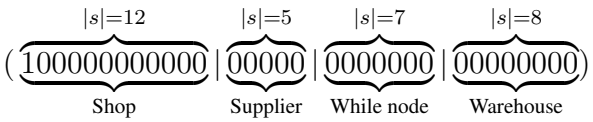

Figure 4.6: The Initial State of the e-shopping system

Details of the execution semantics of the Extended Workflow Graph (EWFG) will be discussed in the next section, whereby further clarification will be given as to the need for defining the additional functions s^I and s^w.

4.4.2 Semantics of the Extended Workflow Graph

If we were to suppose that s_i and s_{i+1} are two states of an Extended Workflow Graph. We would write $s_i \xrightarrow{n} s_{i+1}$ to denote the firing of a node n which would alter the state of s_i to s_{i+1}.

In Section 2.5 the firing rules of the common nodes of the conventional Workflow Graph such as Merge, Activity, Decision, Fork, and Join have also been defined. In addition for Start and Stop nodes of the root Workflow Graph, the firing rules have also been explained. The same set of firing rules could also be applied to these nodes in the EWFG. When these nodes fire, obviously the value of coordinates corresponding to the internal behaviour of Invocation and While nodes remain unchanged (i.e. $s^I_{i+1} = s^I_i$ and $s^w_{i+1} = s^w_i$).

The remainder of this section will discuss the firing rules of While and Invocation nodes. In such nodes, the tokens must move into the children node in order to execute the internal behaviour of the parent node. This is achieved by the firing of the Start nodes of the associated children node. Similarly, when the execution of the internal Workflow Graph (i.e. the children node is terminated), the token must be removed.

For example, in the case of a While node, either the internal behaviour repeats itself, which would mean a new execution of the Start node, or the execution of the While node would be terminated, resulting in the removal of a token from the child Workflow Graph.

Consequently, as a part of the description of the firing rules for the execution of While and Invocation nodes; the execution for their Start and Stop node will be described. This is considered to be different from the execution rules associated with the Start and the Stop node of the root of the EWFG.

Change of States for While

If we were to assume that $s_i = (s_i^E, s_i^I, s_i^w)$ and $s_{i+1} = (s_{i+1}^E, s_{i+1}^I, s_{i+1}^w)$ with $s_i \xrightarrow{n} s_{i+1}$. We would suppose n **is a While node** such that $(n, G) \in \beta$ (i.e. G is the internal behaviour associated to the node n). The While node n is enabled when it has one token in its input edge, as shown in Figure 4.7a. The firing of n removes one token from its input edge and adds one token to the Start node of its associated Workflow Graph $\beta(n).Start$; the While node n is marked by a "1" token to indicate that the Workflow Graph associated to n is executing (i.e. $s_i^w(n) = 1$) as shown in Figure 4.7b.

- n is enabled if $s_i(e) > 0$ for $e \in In(n)$ and subsequently:

$$s_{i+1}^E(e) = \begin{cases} s_i^E(e) - 1 & \text{if } e \in In(n) \\ s_i^E(e) + 1 & \text{if } e \in Out(\beta(n).Start) \\ s_i^E & \text{otherwise.} \end{cases}$$

  The coordinates related to the Invocation nodes remain unchanged (i.e. $s_{i+1}^I = s_i^I$).

  For the coordinates related to the While nodes:

$$s_{i+1}^w(m) = \begin{cases} 1 & \text{if } m = n \\ s_i^w & \text{if } m \neq n. \end{cases}$$

- The firing rule for the Stop node of a child node of a While node can be described in the following manner: if m is $\beta(n).stop$ whereby n is a While node; m is enabled if $s_i(e) > 0$ for $e \in In(m)$ as shown in Figure 4.7c. If the While node should be repeated,

the firing m removes one token from its input edge and adds one token to the Start Node (i.e. $\beta(n).Start$). Otherwise, the firing of m removes one token from its input edge and adds one token to the output edge of the While node n. The token held on the While node n is removed, and the coordinate corresponding to n in the state is marked by "0" to indicate that the process on the Workflow Graph associated to n has been completed, as shown in Figure 4.7d.

$$s^E_{i+1} = \begin{cases} s^E_i(e) + 1 & \text{Repeat } \& \ e \in Out(\beta(n).Start) \\ s^E_i(e) + 1 & \text{No Repeat } \& \ e \in Out(n) \\ s^E_i(e) - 1 & \text{if } \ e \in In(m) \\ s^E_i & \text{otherwise} \end{cases}$$

The coordinates related to the Invocation nodes remain unchanged (i.e. $s^I_{i+1} = s^I_i$).

Whilst for the coordinates related to the While nodes:

$$s^w_{i+1}(m) = \begin{cases} 0 & \text{if } \ m = n \\ s^w_i & \text{if } \ m \neq n \end{cases}$$

Change of States for Invocation

If we were to suppose n is an Invocation node such that $(n, G) \in \beta$ where G represents the internal behaviour that should be performed when n gets executed. The execution of an Invocation node can be divided in two ways as either an asynchronous one-way operation or synchronous request/response. The semantics of such executions can be explained in the following manner:

If n **is an Invocation node with a *one-way operation***, the firing of n removes one token from its input edge, adds one token to its output edge and one token to the Start Node of its associated Workflow Graph $\beta(n).Start$. This means that the parent Workflow Graph continues executing, and at the same time the child Workflow Graph starts executing.

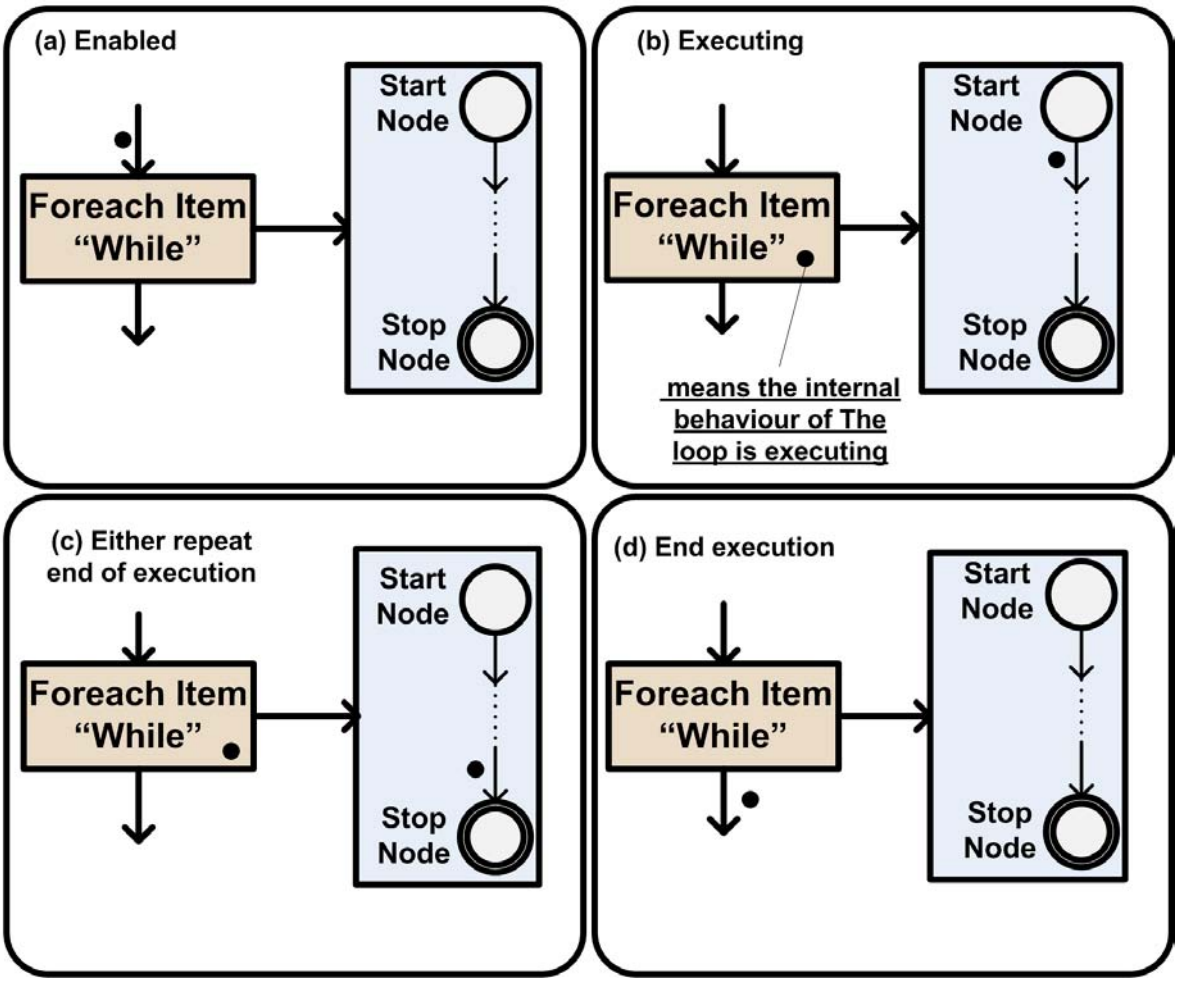

Figure 4.7: While loop structure and semantics

- n is enabled if $s_i(e) > 0$ for $e \in In(n)$ then

$$s^E_{i+1}(e) = \begin{cases} s^E_i(e) - 1 & \text{if } e \in In(n) \\ s^E_i(e) + 1 & \text{if } e \in Out(\beta(n).Start) \\ s^E_i(e) + 1 & \text{if } e \in Out(n) \\ s^E_i & \text{otherwise} \end{cases}$$

For the coordinates related to the Invocation and While nodes remain unchanged and their values are still the same (i.e. $s^I_{i+1} = s^I_i$ and $s^w_{i+1} = s^w_i$).

- The firing rule for the Stop node of a child node of an Invocation node with one-way operation can be explained as follows: if $m = \beta(n).stop$ such that n is an Invocation node, firing m removes one token from its input edge. If there is only one token on the input edge, m is terminated. m is enabled if $s_i(e) > 0$ for $e \in In(m)$ and subsequently:

$$s^E_{i+1}(e) = \begin{cases} s^E_i(e) - 1 & \text{if } e \in In(m) \\ s^E_i & \text{otherwise.} \end{cases}$$

  In this case, the value of Invocations and While coordinates remains unchanged (i.e. $s^I_{i+1} = s^I_i$ and $s^w_{i+1} = s^w_i$).

If n **is an Invocation with a *two-way operation***, the firing of n removes one token from its input edge and adds one token to the Start Node of the Workflow Graph associated to n (i.e. $\beta(n).Start$). Hence, in the two-way operation the execution of the parent Workflow Graph is blocked whilst the execution of the internal behaviour (i.e. the child node terminates). The Invocation node n is marked by a "1" token to indicate that the Workflow Graph associated to n is currently executing (i.e. $s^I_i(n) = 1$).

- n is enabled if $s_i(e) > 0$ for $e \in In(n)$ then

$$s^E_{i+1}(e) = \begin{cases} s^E_i(e) - 1 & \text{if } e \in In(n) \\ s^E_i(e) + 1 & \text{if } e \in Out(\beta(n).Start) \\ s^E_i & \text{otherwise.} \end{cases}$$

  For the coordinates related to Invocation nodes s^I_{i+1}:

$$s^I_{i+1}(m) = \begin{cases} 1 & \text{if } m = n \\ s^I_i & \text{if } m \neq n. \end{cases}$$

  However, the coordinates related to While nodes remain unchanged (i.e. $s^w_{i+1} = s^w_i$).

- the firing rule for the Stop node of a child node of a two-way operation can be described as follows: if $m = \beta(n).stop$ where n is an Invocation node, the firing of m removes one token from its input edge and adds one token to the output edge of the Invocation node n. The token in the Invocation node (i.e. n) is removed and n is marked by "0" to indicate that the process of executing the Workflow Graph associated to n has been completed. m is enabled if $s_i(e) > 0$ for $e \in In(m)$ then

$$s^E_{i+1}(e) = \begin{cases} s^E_i(e) - 1 & \text{if } e \in In(m) \\ s^E_i(e) + 1 & \text{if } e \in Out(n) \\ s^E_i & \text{otherwise} \end{cases}$$

For the coordinates related to Invocation nodes:

$$s^I_{i+1}(m) = \begin{cases} 0 & \text{if } m = n \\ s^I_i & \text{if } m \neq n \end{cases}$$

However, the coordinates related to While nodes remain unchanged i.e. ($s^w_{i+1} = s^w_i$).

Remark 1 *Vanhatalo et al. [128] only assign tokens to the edges of a Workflow Graph. In this thesis, the tokens may be assigned to the edges and While nodes. This means that we are deviating away from Vanhatalo et al. [128] by assigning tokens to While nodes. The decision to allow assignment of tokens to While nodes was not taken lightly and various options were investigated. The key challenge is to ensure that the marking reflects a state of "executing" for the While loop. For example, one may naively suggest that after a While loop is enabled, i.e. the edge prior to the node is marked, and when the While loop executes, the token should be removed from the edge and moved to the child Workflow Graph. However, this may cause ambiguity if a new token arrives at the input edge of n. In this case, a new execution of the internal Workflow Graph may be repeated. This would result in the wrong execution of the Workflow Graph such as execution of a Stop node, which would imply that execution has been*

terminated, whilst there are still tokens in the Workflow Graph.

This section will be concluded by extending the definition of execution traces from Petri net to the EWFG. This will be used in the following chapter when the Diagnosability theory of DES to the Extended Workflow Graph is adopted.

Definition 4.4.6 *Suppose that* s_0 *is a state with* $s_0(e) = 1$ *if* $e \in Out(root.start)$ *(i.e.* e *is an edge out of the Start node of the root) and* $s_0(e') = 0$ *for all other coordinates. We refer to* s_0 *as an initial state. Figure 4.6 represents the initial state for the example in Section 3.1. Any sequence* $s_0 \xrightarrow{n_1} s_1 \ldots \xrightarrow{n_k} s_k$ *is called an Execution Sequence of the Extended Workflow Graph. Occasionally, if there is no chance of ambiguity, we write* $s_0 \xrightarrow{n_1 \ldots n_k} s_k$ *to denote the execution sequence. The set of all reachable states of* T *are defined as* $Reach(T) = \{s_k | \exists n_1 \ldots n_k$ *so that* $s_0 \xrightarrow{n_1 \ldots n_k} s_k\}$*. We can also define the language of an Extended Workflow Graph* T *as* $L(T) = \{n_1 \ldots n_k | \exists s_k \in Reach(T)$ *so that* $s_0 \xrightarrow{n_1 \ldots n_k} s_k\}$*.*

CHAPTER 5

A MODEL-BASED APPROACH TO FAULT DIAGNOSIS

Chapter 4 presents a modelling approach based on adapting the conventional Workflow Graph which is further extended to represent some complex behaviours of BPEL, such as invocation and repetitive scenarios. Our extension is called the Extended Workflow Graph (EWFG). In this chapter, a model-based approach is proposed to adopt the Diagnosability theory of DES to the Extended Workflow Graph (EWFG). In particular, Section 5.1 describes how to augment the EWFG with new information in order to deal with Observability which is considered an essential requirement in the adoption of the DES techniques. Two algorithms are thus introduced to automate the creation of the Diagnoser. Section 5.3 presents the first algorithm which is used to create the Coverability Graph. Such a graph extends the idea of the Petri net Coverability Graph [57], which is explained in Section 2.4.1, in order to capture the entire behaviour of the system in one model. In Section 5.3.1, the Coverability Graph is used to prove that the language produced by the EWFG is a regular language. In other words, the language underlying models supported by these tools, such as BPEL, is considered a regular language. This means that the Diagnosability methods of DES for creating Diagnosers can be applied and adopted in this context, as explained in Section 2.3.5. Consequently, our approach furthers the method

suggested by Giua and Seatzu [57, 58] and Genc and Lafortune [55] in designing the Diagnoser for Petri net models. In Section 5.4, a second algorithm is presented to compute the Diagnoser Coverability Graph.

5.1 Modelling Observability and failure of EWFG

Assume that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG), as defined in Section 4.4. For each $G_i \in \mathscr{V}$, $G_i = (N_i, E_i)$ where N_i is the set of nodes and E_i is the set of edges of G_i. Suppose that $N = \bigcup N_i$ is the set of all nodes of the EWFG. Following the lead of DES, see Section 2.3.4, we partitioned N into two disjoint subsets: *observable nodes* N_{obs} (i.e. their occurrence can be observed) and *unobservable nodes* N_{uo}. This means $N = N_{obs} \cup N_{uo}$ and $N_{obs} \cap N_{uo} = \phi$. For example, if we assume that the node called *Send Item to the Supplier* in the Shop Workflow Graph of the example presented in Section 3.1 is an observable action, as its occurrence can be seen by the Supplier service; whereas the node called *Calculate the Cost* is considered as unobservable node, as the calculation is carried out locally.

Example 5.1.1 *The observability concepts of a Workflow Graph can be applied to the example presented in Section 3.1. The set of the observable and unobservable nodes of the Workflow Graph of the Supplier service of Figure 4.5 is defined:*
N =*{Start Node, Receive Items List, While, Return Items List to Shop, Stop Node}, where* N_{obs}=*{Receive Items List, Return Items List to Shop}, and* N_{uo}=*{Start Node, While, Stop Node}*

Some of the Workflow Graph activities such as Fork, Join, Merge and Decision are used across all Workflow Graphs and the system. We shall therefore distinguish them from the events which directly represent the system events.

Definition 5.1.2 *Internal Actions N_{int} represents the internal actions such as the Start node, Stop node, Fork, Join, Decision, Merge and While in which their execution is performed internally and hence unobservable (i.e. $N_{int} \subset N_{uo}$).*

As explaind in Section 2.3.4, a system may have different types of failure. We therefore write $N_f = N_{f_1} \cup \cdots \cup N_{f_\ell}$ to classify the sets of failure into different categories. All failure nodes are considered unobservable $N_f \subseteq N_{uo}$. For example, some of the scenarios of the execution in the given example can result in a violation of failures. In this context, we have selected two types of failures to demonstrate our approach (i.e. $N_f = \{N_{f_1}, N_{f_2}\}$) which are explained in the following manner:

- A failure N_{f_1} occurs when the customer makes a mistake whilst placing their order. This failure is caused by a data acquisition which may result in either billing of the wrong items or billing of the wrong number of items.

- A failure N_{f_2} represents the case when the Supplier service does not return the same list of items to the Shop service i.e. one or more items are missing from the list. In other words, N_{f_2} is a failure, which may occur after the execution of *Return Items List to Shop* node in Supplier service.

5.2 Diagnosis of the Extended Workflow Graph

This section presents an approach to the diagnosis of a set of business processes whereby each business process is modelled as a Workflow Graph with Invocation and While nodes (WFGIW). The interactions between these Workflow Graphs are captured by tokens that can move from one WFGIW model to another, as explained in Chapter 4. The occurrence of the observable actions of these WFGIW are the only actions that can be seen by other services, including the monitoring tools (i.e. the Diagnoser service in our approach) as shown in Section 2.3.5.

In the Extended Workflow Graph, the failures are explicitly modelled as unobservable actions and the system observation is considered via a subset of actions whose occurrence is observable (i.e. firing an observable event is recognised by the Diagnoser). The diagnosis result is subsequently obtained in two steps. Firstly, the observable behaviour of the system is derived from a set of legal traces of the Extended Workflow Graph. Secondly, these traces are checked

to ascertain whether they include failures by exploiting the explicit model of the system and the observable traces in order to derive what failures have occurred in the system. Hence, the Diagnoser makes use of the system model and the details of the observation to determine the failures, as explained in Section 2.3.5.

Computing the Diagnosability of the Extended Workflow Graph, which is based on adopting the Diagnosability theory of DES, is achieved in the following manner: firstly, we should compute the *Product* of the system, which is the standard way of capturing the entire behaviour of the system in one model. Therefore, we can adopt the Coverability Graph of Petri nets [34] to capture such a *Product*.

In Section 5.3, we shall introduce the concepts associated with the Coverability Graph of an Extended Workflow Graph. We have developed an algorithm to compute the Coverability Graph for any Extended Workflow Graph. Finally, an algorithm to compute the Diagnoser of the system is proposed.

5.3 The Coverability Graph of an Extended Workflow Graph

In Petri nets, the Coverability Graph is used to analyse unbounded nets [113], where it is possible to have infinite number of states. This is considered possible when the number of tokens on the *places* grow infinitely. Therefore, the Coverability Graph is a technique used to obtain a finite graph including all permissible reachable traces in order to solve many issues of Petri nets [57, 58, 55, 34]. This is achieved by introducing the symbol ω, which can be though as "infinity" [96]. It has the property that for each integer $n, \omega > n, \omega \pm n = \omega$.

The Coverability Graph of a Petri net has been widely used to study the dynamic properties of systems [57, 58, 55]. Guia [57] proposed an approach based on the use of the Coverability Graph to address the problem of estimating the marking of a place/transition based on event observation. In [58], they benefited from [57] to present an efficient approach for the fault detection of discrete event systems using Petri nets.

Genc and Lafortune [55] dealt with the diagnosis of failure in Petri net models by extending the Diagnosability of Discrete Event System for systems modelled by Automaton proposed in [118]. Whilst Cabasino et al. [34] study the Diagnosability of Petri nets using the analysis of the Coverability Graph.

In common with their approaches, we are using the Coverability Graph in order to adopt the Diagnosability theory of DES to SoA. In this research, we have extended the idea of the Coverability Graph from Petri nets to the Extended Workflow Graph with Invocation and While nodes, which is presented in Section 4.4. The following definition of a Coverability Graph is a direct extension of a similar definition in Petri nets.

Definition 5.3.1 *A Coverability Graph of an Extended Workflow Graph* $T = (\mathscr{V}, \Sigma)$ *is a graph* $G_{cov} = (N_{cov}, E_{cov})$ *such that:*

I- Each node of the Coverability Graph is marked by a $k-$*dimension vector of coordinates* $\mathbb{N} \bigcup \{\omega\}$*, where* k *is the number of coordinates in the state of an EWFG as defined in Definition 4.4.1.*

II- Each edge of the Coverability Graph is marked by a node of T *(i.e.* $E_{cov} \subseteq N$ *where* N *is the set of all nodes of* T*).*

III- For each reachable set of states $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$*, there is a path* $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_k} \alpha_k$ *such that* $s_i \leqslant \alpha_k$ *for* $1 \leqslant i \leqslant r$*, where* $\leqslant$ *is coordinate ordering of vector in* $\mathbb{N} \bigcup \{\omega\}$*, in which* $\forall n \in \mathbb{N}, n \leq \omega$*.*

In this section, we present an algorithm for the computation of the Coverability Graph which complies with the above definition. Algorithm 2 is proposed as a direct extension of the Coverability Graph Algorithm of Petri nets [96, 113]. In this algorithm, for any given Extended Workflow Graph, from the initial marking s_0, it is possible to obtain a number of new markings for each enabled activity. At each new marking, we can again obtain the next markings. This

ends up in a tree of all possible markings, which is made into a graph by merging identical nodes.

Algorithm 2 The Computation of the Coverability Graph

Output $G_{cov} = (N_{cov}, E_{cov})$
Create an initial node $(1 \ldots 0| \ldots |0 \ldots 0)$
Label the initial node as the root and tag it as "new".
while a node marked by "new" in the Coverability Graph exists **do**
  Select a node marked by "new" α
  if α is identical with a node on a path from the initial node to α **then**
    tag α as "old"
  else
    if no activities are enabled at α **then**
      tag α as "dead"
    else
      for all activities n_i enabled at α **do**
        compute the Marking α' that results from firing n_i at α. The firing rules which are described in Section 2.5 and 4.4 must be extended by $\forall n \; n + \omega = \omega + n = \omega$
        On the path from the root to α if there exists a Coverability Graph node such that $\alpha'' \leqslant \alpha'$ and $\alpha' \neq \alpha''$ i.e. α'' is covered by α', then replace $\alpha'(e) = \omega$ for each e such that $\alpha'(e) > \alpha''(e)$
        Introduce the new α' as Coverability Graph node
        Draw an arc with label n_i from α to α'
        Tag α' "new"
      end for
      Tag α "old"
    end if
  end if
end while

Lemma 5.3.2 *Algorithm 2 produces the Coverability Graph for any given Extended Workflow Graph.*

Proof. We prove this result by induction as follows: for each reachable set of state $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$ there is a path $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} \alpha_r$ such that

for each i $\quad s_i \leqslant \alpha_i, \qquad 1 \leqslant i \leqslant r \qquad$ (*).

Basis step: $r = 0$, s_0, $\alpha_0 = s_\alpha$, (*) is automatically correct. Assume for each r the statement is true. Then, we show the statement to be true for $r+1$ (induction step). Suppose a set of reachable

states $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r \xrightarrow{n_{r+1}} s_{r+1}$. We want to show that there are $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} \alpha_r \xrightarrow{n_{r+1}} \alpha_{r+1}$ such that for $1 \leqslant i \leqslant r+1, s_i \leqslant \alpha_i$. Since $s_0 \xrightarrow{n_1} s_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} s_r$ is of the length r and the statement is true for r. There are $\alpha_1 \ldots \alpha_r$, $\alpha_0 \xrightarrow{n_1} \alpha_1 \xrightarrow{n_2} \ldots \xrightarrow{n_r} \alpha_r$ where for all $1 \leqslant i \leqslant r, s_i \leqslant \alpha_i$. We have to show that there is α_{r+1} such that $\alpha_r \xrightarrow{n_{r+1}} \alpha_{r+1}$ in which $s_{i+1} \leqslant \alpha_{r+1}$. Because $\alpha_r \geqslant s_r$: transition n_r can fire under α_r. This is because edges which inputs to n_r have at least as many tokens as s_r in α_r. Suppose that n_r fires under α_r. We have $\alpha_r \xrightarrow{n_r} \alpha$. α_{r+1} is made out of α by replacing some of coordinates with ω. So, $\alpha \leqslant \alpha_{r+1}$. We will show that $s_{r+1} \leqslant \alpha$ hence $s_{r+1} \leqslant \alpha_{r+1}$. Then, there are the following cases:

Case 1: suppose e is an edge which is not input or output of n_r. So, the number of token on such edge does not change. $s_r(e) = s_{r+1}(e)$. Similarly because $\alpha_r \xrightarrow{n_{r+1}} \alpha_{r+1}$, $\alpha_r(e) = \alpha_{r+1}(e)$. Since $\alpha_r \geqslant s_r$, we have $\alpha_{r+1}(e) = \alpha_r(e) \geqslant s_r(e) = s_{r+1}(e)$. So, $\alpha_{r+1}(e) \geqslant s_{r+1}(e)$ for all e not input or output of n_r.

Case 2: Suppose e is an edge input of n_r. By the firing rules $s_{r+1}(e) = s_r(e) - 1$, $\alpha_r(e)$ can be a number or ω. If $\alpha_r(e)$ is a number, $\alpha_{r+1}(e) = \alpha_r(e) - 1$. So, $\alpha_{r+1}(e) = \alpha_r(e) - 1 \geqslant s_r(e) - 1 = s_{r+1}(e)$. If $\alpha_r(e) = \omega$, $\alpha_{r+1}(e) = \omega$ such that $\alpha_{r+1}(e) = \omega \geqslant s_{r+1}(e)$.

Case 3: Suppose e is an edge input of n_r. By the firing rules $s_{r+1}(e) = s_r(e) + 1$, $\alpha_r(e)$ can be a number or ω. If $\alpha_r(e)$ is a number, $\alpha_{r+1}(e) = \alpha_r(e) + 1$. So, $\alpha_{r+1}(e) = \alpha_r(e) + 1 \geqslant s_r(e) + 1 = s_{r+1}(e)$. If $\alpha_r(e) = \omega$, $\alpha_{r+1}(e) = \omega$ such that $\alpha_{r+1}(e) = \omega \geqslant s_{r+1}(e)$. □

Example 5.3.3 *Applying Algorithm 2 to the example of Section 3.1, A Coverability Graph is produced as shown in Figure 5.1.*

5.3.1 Production of a Regular Language model

In the context of Petri nets, the Coverability Graph is a labelled directed finite graph used to represent the behaviour of a system which involves an infinite number of states. However, if the set of reachable states of a Coverability Graph of a Petri net is finite, this Coverability Graph is the same as the Reachability Graph. As explained in Section 2.4, the Reachability Graph is

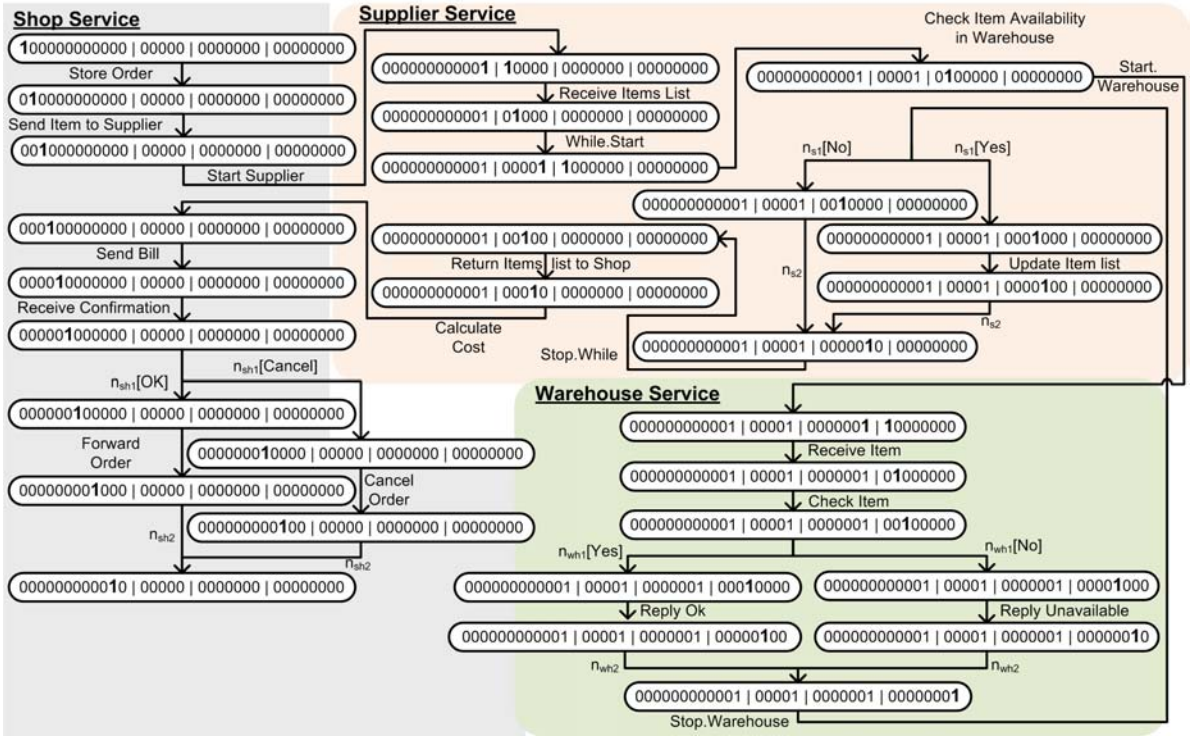

Figure 5.1: The Coverability Graph of the e-shopping example

very similar to the Coverability Graph but the number of tokens in each place is bounded which means there is no chance of having an infinite number of tokens (i.e. the Reachability Graph does not have ω).

In the representation of the Extended Workflow Graph introduced in Section 4.4, the repetitive behaviour is modelled in a *controlled* manner using While loops. In this section we will show that the Extended Workflow Graph, although it includes infinite behaviour; the set of reachable states for them is finite. Intuitively, we will show that the set of reachable states of the Extended Workflow Graph is a subset of the cartesian product of the set of all reachable states of the Workflow Graph produced from "Stripping" each WFGIW at the node of the tree from any possible Invocation or While nodes. First we will start by "Stripping" each node of the tree from all Invocation and While nodes to create a conventional Workflow Graph without any *structured loops*.

Notation. If we suppose that $G = (N, E)$ is a Workflow Graph with Invocation and While nodes. Let us denote by $\widehat{G} = (\widehat{N}, \widehat{E})$ that a Workflow Graph is created by replacing each Invocation and While nodes with a "dummy" activity node with the same name. □

Lemma 5.3.4 *If we suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG) where $\mathscr{V} = \{G_1, G_2 \ldots, G_n\}$. Subsequently for each i, $\pi_i(Reach(T)) \subseteq Reach(\widehat{G_i}) \bigcup \{\overrightarrow{0}\}$ where π_i is the projection of the state vector of T to edge of G_i, $\overrightarrow{0} = (0, \ldots, 0)$ is zero vector of dimension of $\mid E_i \mid$, where E_i is the set of the edges of G_i.*

Proof. The proof is by induction on r where $\sigma = s_0 \xrightarrow{n_1} s_1 \ldots \xrightarrow{n_r} s_r$ is an execution sequence of T and s_r is the r-th reachable state. The first step of induction is trivial as if G_i is the root of the Graph, $\pi_i(s_0)$ would be $(1, 0, \ldots, 0)$, otherwise $\pi_i(s_0) = \overrightarrow{0}$. Suppose that for $0 \leqslant j \leqslant r$ $\pi_i(s_0), \ldots, \pi_i(s_{r-1}) \in Reach(\widehat{G_i}) \bigcup \{\overrightarrow{0}\}$, we must show that $\pi_i(s_r) \in Reach(\widehat{G_i}) \bigcup \{\overrightarrow{0}\}$. In $s_{r-1} \xrightarrow{n_r} s_r$ if n_r is not an edge of G_i then $\pi_i(s_r) = \pi_i(s_{r-1})$ and there is nothing to prove. Similarly if n_r is not a While or Invocation node, the change of state of the overall system and

change of states of $\widehat{G}$ are identical. Therefore, the nontrivial cases are when n_r is a While or an Invocation node. The idea of the proof is that the execution of a While or Invocation node results in the executing of a child Workflow Graph which makes no changes to the state of $\widehat{G_i}$.□

Lemma 5.3.5 *Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG) where $\mathscr{V} = \{G_1, G_2 \ldots, G_n\}$; and suppose that none of the G_i have structured loops, then the set of reachable states of T is a finite set.*

Proof. From lemma 5.3.4 we can show that:
$Reach(T) \subseteq (Reach(\widehat{G_1}) \bigcup \{\overrightarrow{0}\}) \times \cdots \times (Reach(\widehat{G_n}) \bigcup \{\overrightarrow{0}\})$ as each coordinate of a reachable state belongs to a coordinate of one of $Reach(\widehat{G_1}) \ldots Reach(\widehat{G_n})$. If G_i has no structured loop, then $\widehat{G_i}$ has no structured loop. Creating $\widehat{G_i}$ does not modify the topology of G, whilst replacing Invocation and While nodes with activity nodes. Since $\widehat{G_i}$ has no structured loops, $Reach(\widehat{G_i})$ is finite. Hence $Reach(\widehat{G_1}) \ldots Reach(\widehat{G_n})$ are all finite. As a result, $Reach(T)$ is finite □

Theorem 5.3.6 *Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG) where $\mathscr{V} = \{G_1, G_2 \ldots, G_n\}$; and suppose that none of the G_i have unstructured loops, then the language of T is a regular language.*

Proof. Since the set of all reachable states is finite, the Coverability Graph captures all possible reachable states as an Automata. So, the language of T is regular. □

As we have demonstrated that the language produced by an Extended Workflow Graph is a regular model, the Diagnosability methods of creating the Diagnoser of DES, explained in Section 2.3.5, can be adopted and applied in this context.

5.4 The Diagnoser of the Extended Workflow Graph

Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph (EWFG). In the previous section, we showed that the Coverability Graph of an Extended Workflow Graph is the same as the Reacha-

bility Graph. Therefore, the set of all reachable state in the Coverability Graph is the same a the reachable state in the EWFG. Hence, we developed the Diagnoser for the Coverability Graph.

In this section, we shall introduce the *Diagnoser Coverability Graph (DCG)* which is inspired by the definition of the Diagnosability theory introduced in [118, 55]. The Diagnoser Coverability Graph (DCG) is a labelled Coverability Graph built from the Coverability Graph of the system described in Section 5.3. DCG derives the observable behaviour of the system in order to perform an online diagnosis, as explained in Section 2.3.5.

Definition 5.4.1 *The Diagnoser Coverability Graph (DCG) of an Extended Workflow Graph* $T = (\mathscr{V}, \Sigma)$ *is a graph:*

$$G_d = (N_d, E_d, \Delta_f)$$

where:

N_d *is the set of nodes, which are the resulting subset of the observable edges composed of the states of the Diagnoser reachable from the initial state under* σ_d*.*

E_d *is the set of the observable edges. Each edge of the Diagnoser Coverability Graph is marked by an observable node of* T *(i.e.* $E_d \subseteq N$ *where* N *is the set of all nodes of* T*).*

The transition function of the Diagnoser is defined in a similar manner to the transition function of the Coverability Graph; but it has an additional aspect to attach failure labels to the states. These labels are propagated from one state to another with the help of the Label Propagation Function, which we shall describe in this chapter.

The set of failure labels is defined as follows: $\Delta_f = \{F_1, F_2, \ldots, F_3\}$

The DCG is an approximation of the behaviour of the EWFG to include *only* the behaviour which is observable. To be precise, for any observable sequence of actions σ, there is a path in the DCG marked by σ starting from the root of the DCG to a node of the DCG which contains the *set of all states* s; so that there is a sequence μ with $s_0 \xrightarrow{\mu} s$ with $P(\mu) = \sigma$. This would indeed provide an approximation of σ, as the node of the DCG which contains s also

includes all states of the DCG which are reached from s via the firing of unobservable events. In [36, 118, 71], this is referred to as the *Unobservable Reach* of s.

Definition 5.4.2 *Suppose that $T = (\mathscr{V}, \Sigma)$ is an Extended Workflow Graph and s is a Reachable state of T. We shall define the Unobservable Reach of s as follows:*

$$UR(s) = \{s_r | s \xrightarrow{n_1} s_1 \ldots \xrightarrow{n_r} s_r, \forall i \; n_i \in N_{uo}\}$$

The Diagnoser estimates the current state of the system after the occurrence of observable events. In general "state" means the state of the system which we have explained in Section 4.4.1; whereas the "Diagnoser State" relates to the state of the Diagnoser. An introduction to the notation of Diagnoser State will be provided.

5.4.1 The Diagnoser State

The Diagnoser State encodes information about failures into approximated states. The encoded information represents the types of failure which can occur when the system arrives at a given state. The Diagnoser State is therefore an extension of the notion of the State of the Extended Workflow Graph presented in Section 4.4.1.

Definition 5.4.3 *Suppose that T is an Extended Workflow Graph. Each State of the Diagnoser of an Extended Workflow Graph is denoted by (α, ϕ) where α is a state of the Extended Workflow Graph T and $\phi \in \{0,1\}^{\ell}$ in which ℓ is the number of categories of failures. Intuitively speaking, if the coordinate i of ϕ is equal to 1, we infer that under the state α, a failure of type N_{f_i} has happened. But, If the coordinate i of ϕ is equal to 0, the system state is "normal".*

We wish to warn the reader that the word "*State*" is used both for referring to the State of the EWFG and also to the Diagnoser States. If there is no chance of confusion, we shall use the phrase "State" for both.

Notation. We will sometimes denote a State (α, ϕ) of the Diagnoser in the following way:

$$\underbrace{(\underbrace{s_1^1 \dots s_{n_1}^1 | s_1^2 \dots s_{n_2}^2 | \dots | s_1^k \dots s_{n_k}^k}_{\text{State of the EWFG}}}_{\alpha} \underbrace{| \underbrace{\phi(1) \dots \phi(l)}_{\text{encoding of failure}})}_{\phi}$$

whereby $(s_1^1 \dots s_{n_1}^1 | s_1^2 \dots s_{n_2}^2 | \dots | s_1^k \dots s_{n_k}^k)$ is already defined in Section 4.4.1, and $\phi = (\phi(1) \dots \phi(l))$ is the encoding of failure in which l is the number of failure categories specified for the system, where for example, $\phi(i) = 1$ if a failure of type N_{f_i} has occurred. □

Example 5.4.4 *Figure 5.2 depicts the initial state of the running example explained in Section 3.1. It has the same structure of the State of the Extended Workflow Graph, but with an additional part for failures. According to our example, this additional part has two coordinates; as we are dealing with two failures. In this state, there is only one token in the first coordinate of the Shop part. Moreover, the system is in a normal status at this state as the failure part does not have any tokens.*

$$(\overbrace{10000000000}^{|s^E|=12}|\overbrace{00000}^{|s^E|=5}|\overbrace{0000000}^{|s^E|=7}|\overbrace{0000000}^{|s^E|=8}|\overbrace{00}^{|N_{f_\ell}|=2})$$

Shop | Supplier | While node | Warehouse | Failures

Figure 5.2: The initial State of example 5.4.4

Each node of the DCG is marked by a set $n_d = \{(\alpha_1, \phi_1), \dots, (\alpha_r, \phi_r)\}$, where $\alpha_1, \dots, \alpha_r$ are reachable states of the Extended Workflow Graph. As explained earlier, $\alpha_1, \dots, \alpha_r$ contains all states of the EWFG obtained from the firing of unobservable transitions (i.e. $\{\alpha_1, \dots, \alpha_r\} = UR(\alpha_1, \dots, \alpha_r)$. To obtain the failure vector coordinates $\phi_1, \dots, \phi_r$, a new function called the Label Propagation Function is required [118]. The function modifies the failure state, in case of failure occurrence and also, as the name suggests, propagates the information about the failure in one state to a subsequent state.

5.4.2 Label Propagation Function (LPF)

Label Propagation Function (LPF) is used to propagate the fault labels consistent with the trace of activities.

Definition 5.4.5 *Suppose that T is an Extended Workflow Graph with the set of Reachable States $Reach(T)$ and the set of nodes N. A Label Propagation Function is a function LP : $Reach(T) \times \{0,1\}^{\ell} \times N \to \{0,1\}^{\ell}$, so that $LP(\alpha, \phi, n) = \phi'$ where the $i-$th coordinate of ϕ' is defined by*

$$\phi'(i) = \begin{cases} 1 & \text{if } n \in N_{f_i} \text{ executes at state } \alpha \\ \phi(i) & otherwise. \end{cases}$$

ϕ' is the encoding of failure for the state which results from the firing of n under the state α. If $\alpha_0 \xrightarrow{n_1} \alpha_1 \ldots \xrightarrow{n_r} \alpha_r$ we will abuse the notation and write $LP(\alpha, \phi, n_1 n_2 \ldots n_r)$ to represent the successive application of LP to $n_1, n_2, \ldots, n_r$. Subsequently we shall require a final piece of notation before presenting an algorithm for creating the Diagnoser Coverability Graph of an Extended Workflow Graphs.

Notation. Suppose that (α_i, ϕ_i) appears in the labelling of one of the nodes of the DCG. We write $\mathbf{F}(\alpha, \phi)$ to denote the set of all (β, ϕ') for which there is a sequence of unobservable events $n_1 n_2 \ldots n_r$ such that $\alpha \xrightarrow{n_1 \ldots n_r} \beta$ and $\phi' = LP(\alpha, \phi, n_1 \ldots n_r)$. □

This means a label of a Diagnoser state is propagated to the next states which are reached via a set of unobservable activities.

5.4.3 Diagnoser Coverability Graph Algorithm

Algorithm 3 provides a systematic procedure which complies with the above definitions in order to compute the Diagnoser Coverability Graph (DCG). The algorithm constructs the DCG starting from the initial node that is labelled by a pair $(\alpha, \mathbf{0})$; where α is the initial node of the Reachability Graph and $\mathbf{0}$ is a vector with coordinates of dimension l, the number of failure

categories. We subsequently consider all the observable activities which are enabled in this state. For each enabled activity, we compute the next reachable markings. If for example n_i is an observable activity, which is enabled at the initial marking, the new marking which results from firing n_i is computed. If the new marking is not involved in any previous nodes, a new node is added to the graph. Consequently, an arc from the initial node to the new node is drawn and labelled by n_i. However, if the new marking already exists in the graph, then it is discarded. This procedure is repeated until considering all possible markings.

Algorithm 3 Creating the Diagnoser Coverability Graph (DCG)

INPUT: Reachability Graph of an EWFG T
OUTPUT: Diagnoser Reachability Graph $G_{DCG} = (N_{DCG}, E_{DCG})$
Create a first node of DCG, mark it "new", and include in it a vector $(\alpha, \mathbf{0})$, where α is the initial node of the Reachability Graph and $\mathbf{0}$ is a vector with coordinates 0 of dimension l, the number of failure categories
while there exists a node of the DCG tagged "new" **do**
  Select a state of the DCG $S = \{(\alpha_1, \phi_1), \ldots, (\alpha_r, \phi_r)\}$ which is tagged by "new"
  Iterate through the list (α_i, ϕ_i) one-by-one
  for all observable actions which are enabled under α_i with $\alpha_i \xrightarrow{n} \lambda$ **do**
    Let $S' := \{(\lambda, LP(\alpha_i, \phi, n))\}$
    $S' := S' \cup \mathbf{F}(S')$
    if S'_i already exists in DCG **then**
      discard it
    else
      Create a State node marked by S' and tag it as "new"
      draw an arc from S to S' marked by n
    end if
    Remove the tag "new" from S after finishing the iteration
  end for
end while

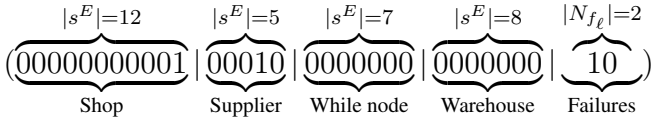

Figure 5.3: An example of the Diagnoser state

Lemma 5.4.6 *Algorithm 3 produces a Diagnoser Coverability Graph (DCG) for any given Extended Workflow Graph.*

Proof. In [33], it has been shown that the Coverability Graph can be viewed as an Automaton. Therefore, the proof here is similar to the proof of producing the Diagnoser for system modelled as an Automaton presented at [118]. □

Example 5.4.7 *Applying Algorithm 3 to the example of Section 3.1, A Diagnoser Coverability Graph (DCG) is produced as shown in Figure 5.4. It can be seen that the states of the Diagnoser are similar to the states of the Coverability except there is new part added to encode the failure information. The failure part has a length of 2 since we assume that there are two types of failures for this example N_{f_1} and N_{f_2} as explained in Section 5.1. For example, Figure 5.3 depicts the system state which has a failure of type N_{f_1}. This state can be reached after executing the node called Return Items to Shop in the Supplier Workflow Graph as shown in Figure 4.5. In the Shop part, a token is assigned to the last coordinate which is associated to the Invocation node to indicate that the Supplier Workflow Graph is executing as an internal action of the Invocation node. A token is also assigned to the 4-th coordinate of the Supplier part which is mapped to the output edge of a node called Return Items list to Shop. The failure part shows that after this execution a failure of type N_{f_1} has occurred.*

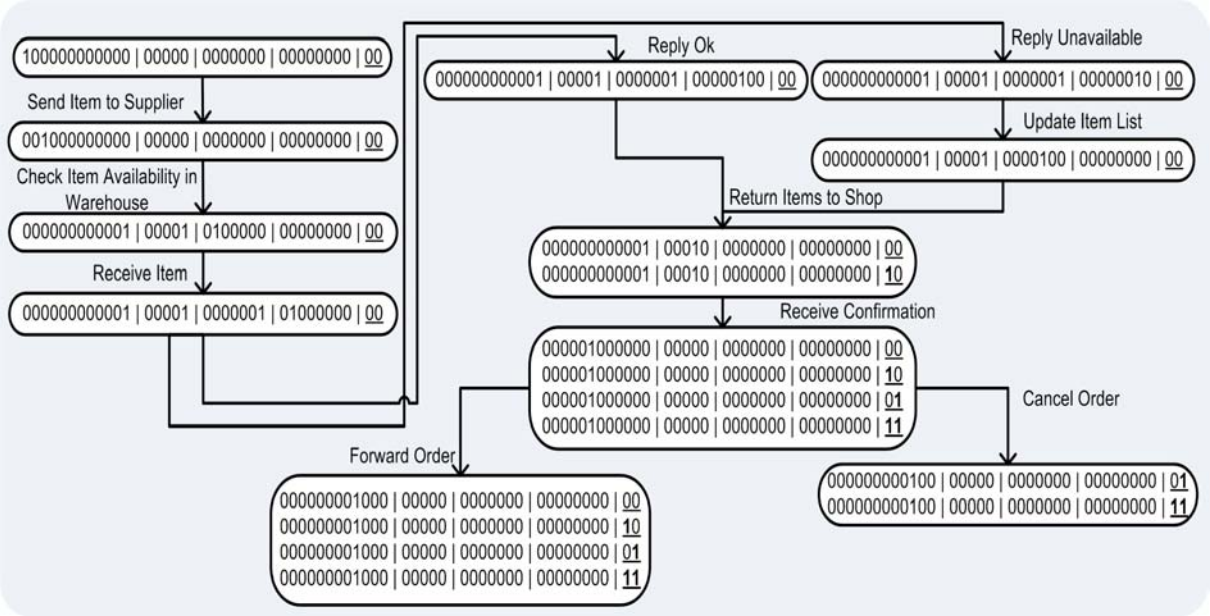

Figure 5.4: The DCG of the Extended Workflow Graphs of the e-shopping example

CHAPTER 6

INTEGRATION OF THE DIAGNOSER

After producing the Diagnoser as explained in the previous chapter, we propose various ways to implement the Diagnoser in SoA. Section 6.1.1 presents a method to implement the produced Diagnoser as a BPEL service interacting with the existing services. Section 6.1.2 presents another method for implementation as a stand-alone Java class deployed as a Web service. Afterwards, we propose two methods to integrate the implemented Diagnoser into the system. Section 6.2.1 presents an option which requires modifying the BPEL files to include extra *Invoke* activity which is used to execute the Diagnoser after each invocation. Section 6.2.2 introduces another option based on the use of a *Protocol Service* to accomplish the interactions between the Diagnoser and the system services. Section 6.3 introduces subsequently the idea of the *Protocol Service*; whilst the automated generation of the *Protocol Service* is represented in section 6.4.

6.1 Implementations of the Diagnoser

In this section, we will present two different methods for the implementation of the produced Diagnoser. There is a choice of implementing the Diagnoser as a BPEL service or as a dedicated Java class deployed as a Web Service. In this section, we shall briefly describe these choices, the details of which will be further described in Chapter 7.

6.1.1 The Diagnoser as a BPEL Service

This method is based on the implementation of the Diagnoser as a BPEL service interacting with existing services. Such a BPEL service includes a *Switch* activity involving a number of *Cases* corresponding to the observable events of the Diagnoser. Each *Case* is used to evaluate the status of the system according to the approximation captured in the Diagnoser states. As a result, the service returns *N* for a normal state or the information related to the occurrence of failures, as described in Chapter 5. In particular, in case of a failure, the type of the failure and its cause will be included in the diagnosis result.

6.1.2 The Diagnoser as a Web Service

This method is based on the implementation of the Diagnoser as a stand-alone Java class deployed as a Web service. This is similar to the previous implementation, and the conditional statements in the form of *if-then-else* are used to represent the behaviour of the Diagnoser.

In order to monitor the behaviour of a set of services, the Diagnoser, which is implemented by one of these methods, should incorporated into the system. In the following section, we will describe two possible methods for integrating the implemented Diagnoser into the system.

6.2 Integrating the Diagnoser

This section will propose two methods for integrating the implemented Diagnoser into the system. The first option is based on adding an extra Invoke activity to execute the Diagnoser after each invocation activity. The second relies on the use of a *Protocol Service*.

6.2.1 Adding extra Invoke Activity

In this method, the implemented Diagnoser is deployed into the server with the existing services. The interaction between these services and the Diagnoser is then accomplished by adding an extra Invoke activity after each invocation. In general, the Invoke activity is used to execute the

services provided by partners via *PartnerLinks* and *portTypes* attributes.

The *PartnerLink* defines the different parties that interact with the business process (i.e. the relationships that a BPEL model will employ in its behaviour) [73]. The *portType* attribute is optional for readability, and its value must match the portType value implied by the PartnerLink and the role implicitly specified by the activity [66]. Therefore, the PartnerLink of the Invoke activity, which is added to execute the Diagnoser, is assigned to the *WSDL* file of the implemented Diagnoser. This interaction with the Diagnoser requires assigning the execution trace to the input variable (*inputVariable*) of the Invoke activity (i.e. the input variable of the Diagnoser). The diagnosing result, which is expected to be received from the Diagnoser, will be captured in an output variable called *outputVariable*. Figure 6.1 depicts an example of an Invoke activity construct used to execute the Diagnoser.

```
<invoke name="ExecuteDiagnoser"
    partnerLink="DiagnoserService"
    portType="emp:DiagnoserService"
    operation="CheckStatus"
    inputVariable="ExecutionTrace"
    outputVariable="DiagnosingResult"
/>
```

Figure 6.1: Invoke Activity to execute the Diagnoser

Example 6.2.1 *Figure 6.2 presents a simple example of how to use an extra Invoke activity to execute the Diagnoser. In this example, there are three services: Shop, Supplier and Warehouse, and consider the Invocation activity called Send items to Supplier of the Shop service. This activity is used to execute the Supplier service. To accomplish this interaction, the Shop service sends a request to the Supplier service and waits for a response. After completing the process in the Supplier service, the invocation result will be returned to the invoker (i.e. the Shop*

service). Finally, the Shop service interacts with the Diagnoser to determine the system status after executing the Supplier service.

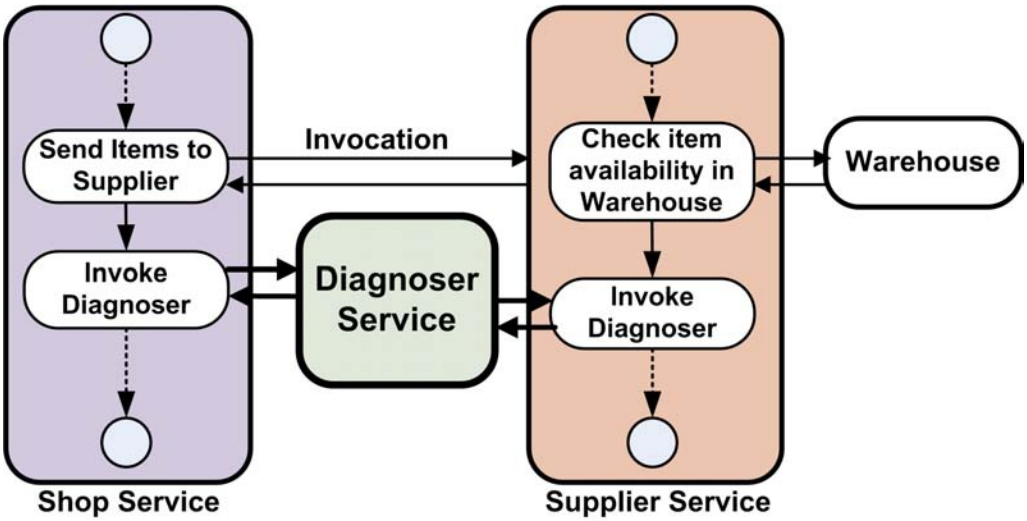

Figure 6.2: Example of adding extra Invocation

In this method, the interaction between the system services and the Diagnoser is considered a choreography architecture [73], see Section 2.1.2. This is because the exchange of messages among parties is based on the collaboration between the system services and the Diagnoser. Moreover, the system services do not rely on a central coordinator to accomplish such interactions.

In general, each service knows exactly when to execute its operations and with whom to interact. All participants of the choreography should know the business process, operations to execute, messages to exchange, and the timing of message exchanges. Hence, this architecture is described from the perspective of all parties. From a composite Web services point of view, the choreography architecture is not flexible, as it requires inclusion of extra Invoke activity to execute the Diagnoser after each invocations, which may lead to extending the size of the BPEL model. Despite this, this architecture is still suitable for small projects, where a small number of services are involved.

6.2.2 Adding a Protocol Service

To overcome the drawbacks which face the choreography architecture, we have proposed an enhanced integration method. In this method, the Diagnoser is produced with a new service called a *Protocol Service*, which is integrated into the system as a service to accomplish the interaction between the system services and the produced Diagnoser.

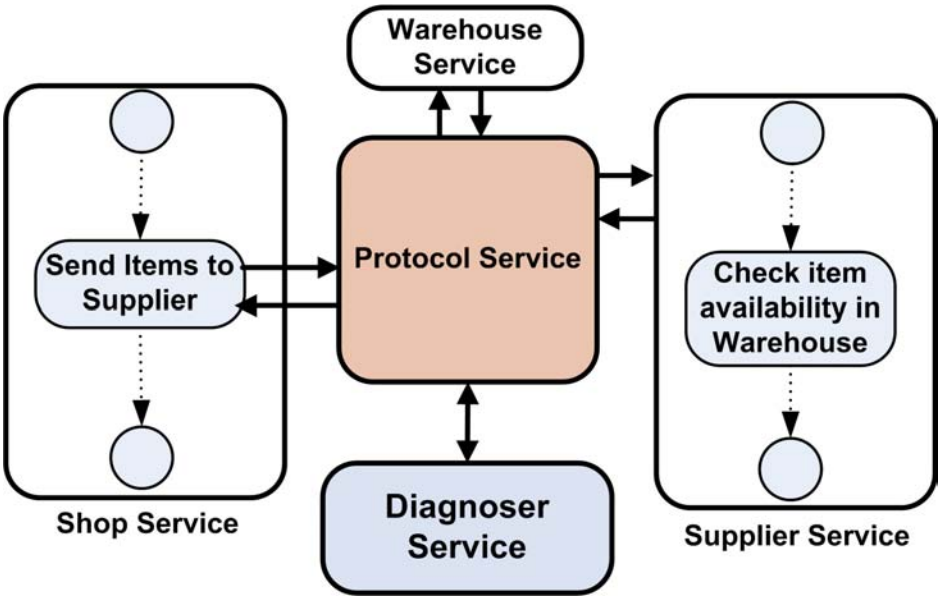

Figure 6.3: Example of using a Protocol Service

The *Protocol Service* carries out the interaction between the system services and the Diagnoser as follows: firstly, the invoker sends an invocation request, which includes the name of the destination service as a variable, to the *Protocol Service*. Based on the request, the service of which its name matches the value of that variable will be executed by the *Protocol Service*. In BPEL, there are two types of invocation [73]. The first is an *one-way operation* which requires only the input variables of the destination and it does not return a response to the invoker (i.e. it is similar to *void* functions in the programming languages). Therefore, the *Protocol Service* sends the request to the destination and proceeding further, does not wait for a response. The *Protocol Service* subsequently executes the Diagnoser to determine the system status after this

execution. Finally, the *Protocol Service* returns the diagnosing result to the invoker.

The second type of invocation is a *two-way operation*, which can be explained similarly to the one-way operation. However, the *Protocol Service* should wait for a response from the destination service before further execution. When the *Protocol Service* receives the response from the destination service, it executes the Diagnoser to identify the system status. Finally, the *Protocol Service* returns the diagnosing and the invocation result obtained from the destination service to the invoker.

Example 6.2.2 *Figure 6.3 presents an example of how to integrate the Protocol Service and the Diagnoser into the system of the running example explained in Section 3.1. This system has three services namely Shop, Supplier and Warehouse.*

In this method, the interaction between the system services and the Diagnoser is coordinated by the *Protocol Service* and is considered as orchestration architecture, where the services interact with each other from the perspective and the control of a single endpoint called *coordinator*, see Figure 6.3. In this architecture, the *Protocol Service* takes the role of a central coordinator controlling how the involved Web services are coordinated and executed. The Web services participating in the orchestration, do not know that they are involved in a composition process, as only the *Protocol Service* is aware of this engagement, with all interactions being accomplished by them. Therefore, the destination services always receive the invocation requests from the *Protocol Service* on behalf of the source services.

From the composing Web services point of view, using the *Protocol Service* is considered a more flexible architecture offering the following advantages over choreography architecture which is based on adding extra Invoke activity [73]:

- The coordination of component processes is centrally managed by a known coordinator.
- Web services can be incorporated without being aware that they are taking part in a business process.

Table 6.1: Different methods of implementation

Integration via \ Diagnoser as	BPEL	Web Service
Adding extra Invocations	Method 1	Method 3
Protocol Service	Method 2	Method 4

- Alternative scenarios can be put into place in case of a fault.

According to the integration and implementation methods, our approach supports four different integration methods, as depicted in Table 6.1. For example, in Method 1, the Diagnoser is implemented as a BPEL file which is integrated into the system by adding extra Invoke activity.

In the following section, we shall describe the *Protocol Service* model and how it is automatically produced and integrated into the system with the help of the MDA.

6.3 The Protocol Service model

The *Protocol Service* acts as a proxy between the BPEL and the target services specified in the original PartnerLinks (PLs) of the BPEL service. The *Protocol Service* is generated as a BPEL service carrying out the following two tasks. Firstly, it performs the interaction between the source and the target services. Secondly, it interacts with the Diagnoser after each invocation to determine the status of the system. Thus, when a BPEL service wishes to interact with another service, it sends a request to the *Protocol Service*. This request includes the details of the destination service and the execution trace of the source service. Then, if the invocation request is one-way operation, which means there is no result expected from the target, the *Protocol Service* sends the request to the target and continues to the next step. Whereas, if the invocation request is a two-way operation, the *Protocol Service* invokes the destination service and waits for a response before proceeding further. This response includes the invocation result and the execution trace of the target. The *Protocol Service* subsequently interacts with the Diagnoser to check whether the system status is in a *normal* state or a failure has occurred. With the help of

the running example the *Protocol Service* will be explained.

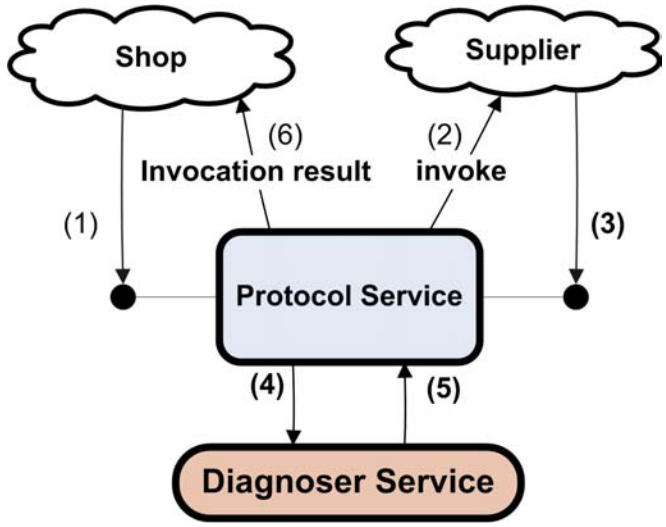

Figure 6.4: A scenario involving the Protocol Service Interaction to identify a failure

Example 6.3.1 *Figure 6.4 represents a scenario involving a simple interaction between the Shop and the Supplier service. The interaction between these two services is carried out through the Protocol Service. If we assume that the Shop service wishes to invoke the Supplier service, then the following six steps (enumerated in the picture) are performed. Firstly, the Shop service sends an invocation request involving the information details to the Protocol Service. These include:*

- *The name of the destination service (i.e. the Supplier service in this example).*
- *The input variables which are required by the destination service, such as the Product ID.*
- *The execution trace of the source service.*

The Protocol Service stores the execution trace of the Shop service in its records, whilst invoking the Supplier service. After completing the execution of the Supplier service, the invocation result and the execution trace of the Supplier service, are returned to the Protocol Service. The Protocol Service then interacts with the Diagnoser. Based on the information of the execution

trace provided by the Protocol Service, the Diagnoser determines the status of the system. Finally, the Protocol Service forwards the diagnosing result received from the Diagnoser and the invocation result obtained from the Supplier service to the Shop service.

6.4 Automated Generation of the Protocol Service

Figure 6.5 depicts an outline of how to automate the creation of the *Protocol Service*. To automatically generate the *Protocol Service*, the following steps are carried out. Firstly, a new BPEL service called "*Protocol Service*" is created. A *Switch* activity, which supports conditional selection based on conditions defined in *Case* elements, is added to this BPEL file. For each service of the system, a Case element is added to the *Switch* activity. In this context, each *Case* includes an *Invoke* activity which is used to execute one of the system services. The *Switch* activity is then followed by an *Invoke* activity used to execute the Diagnoser.

Example 6.4.1 *If we consider the Shop, Supplier and Warehouse services of the running example explained in Section 3.1. Figure 6.6 depicts the Protocol Service which is used to accomplish the interaction between these services and the Diagnoser. This model has a Switch activity involving three Cases: Case 1 to invoke the Shop service, Case 2 to invoke the Supplier service and Case 3 to invoke the Warehouse service. The Switch activity is followed by an Invoke activity to execute the Diagnoser.*

6.4.1 The Transformation rules

Figure 6.7 depicts an outline of the automated creation and integration of the *Protocol Service*. The automated creation of the *Protocol Service* is achieved by defining a number of model transformation rules based on Model Driven Architecture (MDA). This method requires two items:

1. The metamodel of the annotated BPEL as the source and the destination of the model transformation.

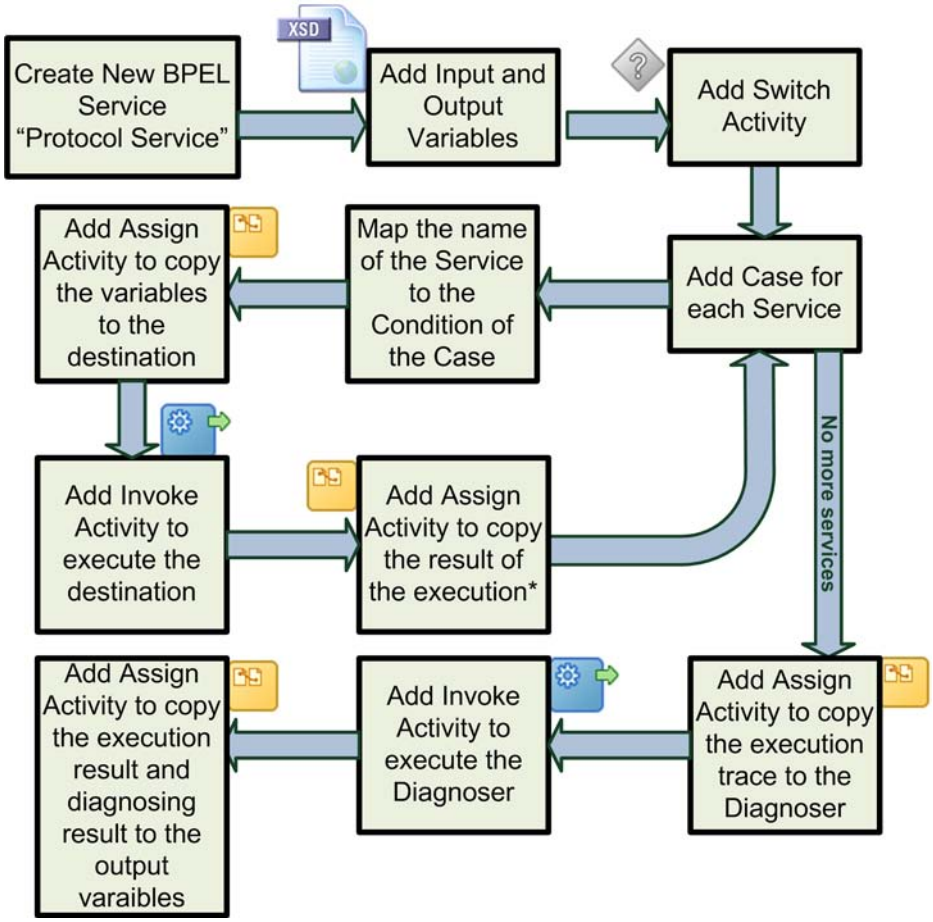

Figure 6.5: An Outline of the Automated Creation of the Protocol Service

2. The transformation rules from the original BPEL models to new BPEL models with the integrated *Protocol Service*.

The metamodel of the annotated BPEL has been presented in Section 3.3. We will now describe the transformation rules which are explained in the following manner: firstly, a new blank BPEL file, called a *Protocol Service*, is created. The input and the output variables of the *Protocol Service* are then generated. The variables of a BPEL file are always declared in the XML Schema Definition (XSD) of the BPEL [124]. XSD is a document structure and type definition specification, which provides the syntax specification and constraints for both structure and content. In this context, three input variables are required to be defined. These include the following:

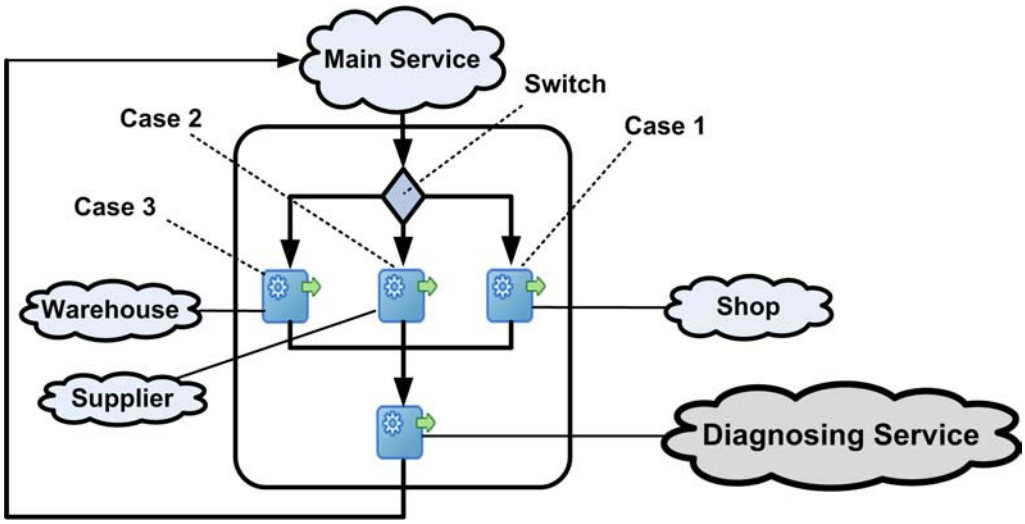

Figure 6.6: An Example of the Protocol Service

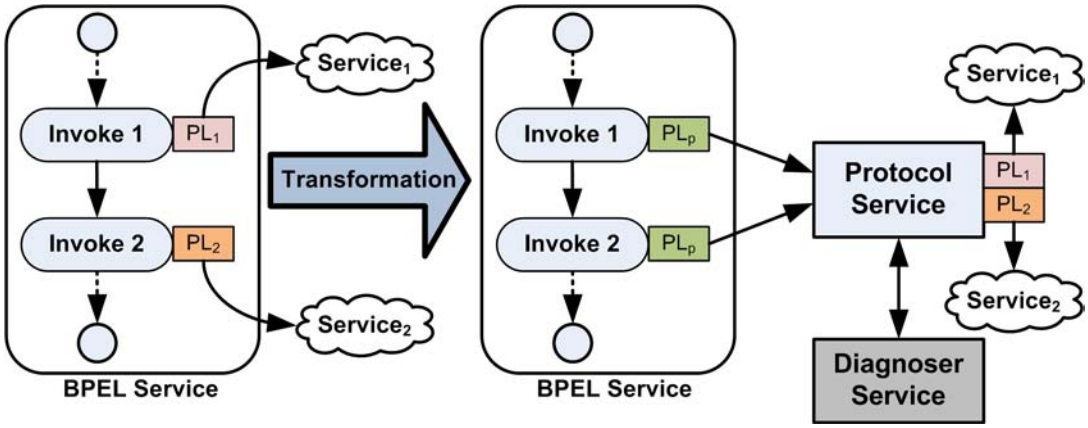

Figure 6.7: Protocol Service generated and embedded into the BPEL process.

- A variable called *service* used to store the name of the destination service.
- A variable called *execution_trace* to keep the execution trace.
- The *destination_inputs* to capture the inputs of the destination service separated by comma.

Figure 6.8 depicts the declarations of these variables in the XML Schema Definition (XSD) file of the *Protocol Service*.

In addition to the input variables, a set of output variables are also generated. Firstly, a variable called *result* is defined to capture the result returned as a response of the destination service.

```
<element name="ProtocolServiceRequest">
        <complexType>
             <sequence>
                 <element name="service" type="string"/>
                 <element name="execution_trace" type="string"/>
                 <element name="destination_inputs" type="string"/>
             </sequence>
        </complexType>
</element>
```

Figure 6.8: Input Variables of the Protocol Service

Secondly, the entire execution trace is captured by an output variable called *execution_trace*. Finally, *system_status* is an output variable used to store the diagnosing result received from the Diagnoser. Figure 6.9 depicts the part of the XML Schema Definition (XSD) file of the *Protocol Service* which includes the declarations of the output variables.

```
<element name="ProtocolServiceResponse">
        <complexType>
            <sequence>
               <element name="result" type="string"/>
               <element name="execution_trace" type="string"/>
               <element name="system_status" type="string"/>
            </sequence>
        </complexType>
</element>
```

Figure 6.9: Output Variables of the Protocol Service

After defining the input and the output variables, a *Switch* activity with multiple *Cases* is added to the BPEL file, where each *Case* is used to execute one of the system services. The name of the *service* is therefore mapped to the value of the boolean expression of that *Case*.

This means if the condition of the *Case* of the *Switch* activity matches the value of the input variable called *service*, then that *Case* will be executed. As a result, the number of the *Cases* is equal to the number of the services involved in the system. In order to execute the service which matches the condition of the *Case*, an *Assign* activity is added to that *Case* in order to copy the execution trace of the system and the input variables from the invoker to the destination service. The *Assign* activity is subsequently followed by an *Invoke* activity used to execute the destination service.

Based on the type of the execution, the Invocation is performed either synchronously or asynchronously. Synchronous Invocations are defined using request-response operations; whilst Asynchronous Invocation can be defined using one-way operations. In the case of the Synchronous Invocation, the *Invoke* activity is followed by an *Assign* activity used to copy the Invocation result from the destination service to the output variables of the *Protocol Service*. The Asynchronous Invocation can be explained similarly, but there is no need to be followed by an *Assign* activity, as there is no response expected from the destination service. In the following an example of how to produce such *Cases* will be presented.

```
<assign name="Assign_ProductID">
 <copy>
  <from variable="ProductIDVar" part="payload"/>
  <to variable="SenditemstoSupplier_InputVariable"
                                    part="parameters"/>
 </copy>
</assign>
<invoke name="SenditemstoSupplier"
partnerLink="SupplierService"
portType="ns1:SupplierService"
operation="process"
inputVariable="InvokeSupplierService_InputVariable"
outputVariable="InvokeSupplierService_OutputVariable"/>
```

Figure 6.10: Example of Invoke Constructor without Protocol Service

Table 6.2: Mapping the result of the Invocation to the Assign Activity

Source	Target
The result obtained from the destination.	The output variable called *result*.
The diagnosing result obtained from the Diagnoser.	The output variable called *system_status*.
The execution trace.	The output variable called *execution_trace*.

Example 6.4.2 *Figure 6.12 depicts an example of how to add a Case to the Switch activity of the Protocol Service in order to execute the Supplier Service. This Case can be executed only when the value of the input variable called service is equal to "Supplier".*

After the inclusion of all possible *Cases* of the *Switch* activity, an *Assign* activity is added to the *Protocol Service* in order to copy the execution trace of the system onto the input variable of the Diagnoser service. The *Assign* activity is subsequently followed by an *Invoke* activity used to execute the Diagnoser synchronously. Finally, an *Assign* activity is added to copy both the diagnosing result obtained from the Diagnoser and the invocation result received from the destination service to the output variables, as shown in Table 6.2.

After producing the *Protocol Service*, each BPEL service is parsed in order to obtain the set of *Invoke* activities. The PartnerLink (LP) of each *Invoke* activity is then replaced by the PartnerLink of the *Protocol Service*. In addition, the *Assign* activity, which precedes the *Invoke* activity, and used to assign the inputs of the invocation activity to the destination service, is modified. Therefore the existing *Copy* operation, which is used to copy the input values onto the variables of the destination service, is adapted to copy the input variables onto the *Protocol Service*. A simple example of how to apply such adoptions has thus been provided.

Example 6.4.3 *Figure 6.10 depicts an Invoke activity used to invoke the Supplier service before modifications are applied to integrate the Protocol Service. Figure 6.11 depicts this Invoke activity after applying the modifications explained above in order to carry out the invocation*

```
<assign name="Assign_ProductID">
 <copy>
  <from expression="SupplierService"/>
  <to variable="SenditemstoSupplier_InputVariable" part="parameters"
      query="/client:ProtocolServiceRequest/client:service_name"/>
 </copy>
  <copy>
   <from variable="ProductIDVar" part="payload"/>
   <to variable="SenditemstoSupplier_InputVariable" part="parameters"
      query="/client:ProtocolServiceRequest/client:productID"/>
  </copy>
</assign>
<invoke name="SenditemstoSupplier"
  partnerLink="ProtocolService"
  portType="ns1:ProtocolService"
  operation="process"
  inputVariable="InvokeProtocolService_InputVariable"
  outputVariable="InvokeProtocolService_OutputVariable"/>
```

Figure 6.11: Example of Invoke Constructor with Protocol Service

through the Protocol Service.

The presented model transformations are implemented in order to automate the creation and integration of the Diagnoser and the Protocol Service. In the following chapter, we shall describe the implementation of our approach which is developed as a Plugin for Oracle JDeveloper.

```
<case condition="bpws:getVariableData('inputVariable',
  'payload','ProtocolServiceRequest/service')='Supplier'">
 <sequence name="Sequence_1">
  <assign name="Assign_ProductID">
    <copy>
     <from variable="inputVariable" part="payload"
       query="ProtocolServiceRequest/destination_inputs"/>
     <to variable="Invoke_Supplier_process_InputVariable"
       query="SupplierServiceProcessRequest/input"/>
     </copy>
  </assign>
  <invoke name="Invoke_SupplierService" partnerLink="SupplierService"
    portType="ns2:SupplierService" operation="process"
    inputVariable="Invoke_SupplierService_process_InputVariable"
    outputVariable="Invoke_SupplierService_process_OutputVariable"/>
  <assign name="Assign_Obtained_Result_to_OutputVar">
    <copy>
      <from variable="Invoke_SupplierService_process_OutputVariable"
         query="SupplierServiceProcessResponse/result"/>
      <to variable="outputVariable" part="payload"
        query="ProtocolServiceResponse/result"/>
    </copy>
    <copy>
     <from variable="Invoke_SupplierService_process_OutputVariable"
         query="SupplierServiceProcessResponse/execution_trace"/>
     <to variable="outputVariable" part="payload"
         query="ProtocolServiceResponse/execution_trace"/>
    </copy>
  </assign>
  </sequence>
</case>
```

Figure 6.12: An example of a Case of the Switch activity of the Protocol Service

CHAPTER 7

IMPLEMENTATION OF A DIAGNOSING FRAMEWORK

This chapter will introduce the implementation of our approach presented in the previous chapters. The approach has been implemented as a plugin for Oracle JDeveloper. Section 7.1 presents the architecture of our implementation.

7.1 Architecture of the Tool

Figure 7.3 depicts an overview of our tool as described in the previous chapters. In this implementation, the Extended Workflow Graph is used as a modelling language for specifying models of business processes. The computation of the Diagnoser has been achieved in the following manner: firstly, the user produces a model of the system in Oracle JDeveloper as BPEL representations. Our tool subsequently extracts an equivalent Extended Workflow Graph for the BPEL files and their XML Schema Definition (XSD). Afterwards, the tool applies the Algorithm 2 explained in Section 5.3 to produce the Coverability Graph. Succeeding this, the Algorithm 3, which is described in Section 5.4, is applied to the Coverability Graph in order to produce the Diagnoser Coverability Graph (DCG). The DCG is then automatically implemented as a Diagnosing Service in SoA as explained in Chapter 6. Finally, the implemented

Diagnoser is automatically integrated and deployed into the system.

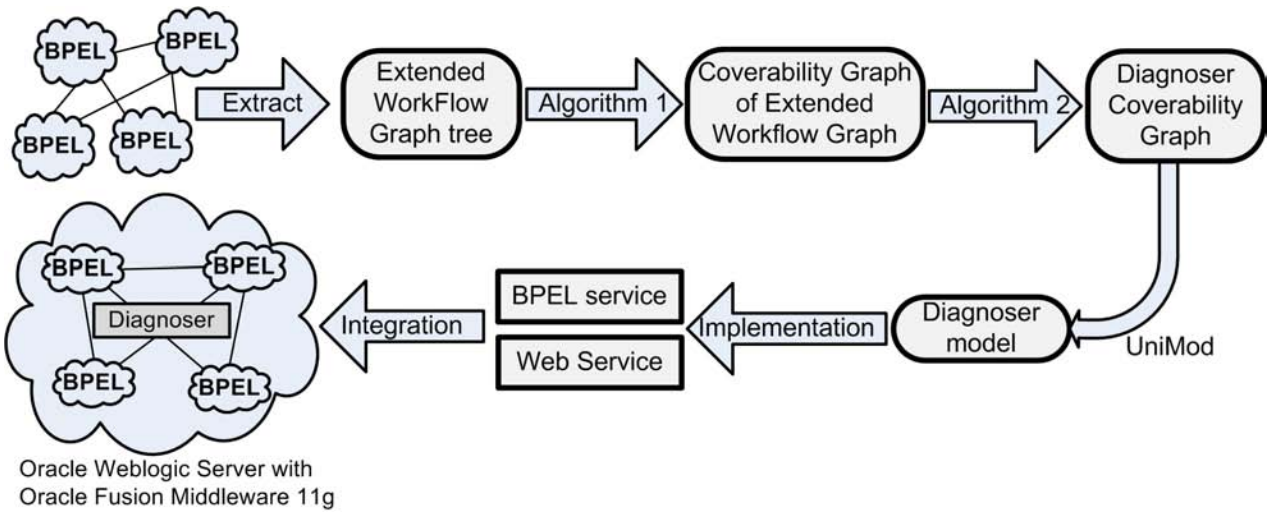

Figure 7.1: Overview of the implementation

Figure 7.2 shows a high level view of the components used by our tool. The tool makes use of the *JDom (Java Document Object Model)* [1] which is an open source library for Java-optimised XML data manipulations. The *JDom* library is used to read the BPEL files of the system and their XML Schema Definition (XSD), in order to extract the Extended Workflow Graph tree as explained in Chapter 4. The tool then applies Algorithm 2 and Algorithm 3 to produce the Diagnoser Coverability Graph (DCG).

The Diagnoser Coverability Graph (DCG) can be automatically implemented and integrated into the system by various possible ways, as explained in Chapter 6. In particular, our tool supports four different integration methods as described in Table 6.1. In general, implementing the Diagnoser in these methods is based on using a Plugin called *UniMod* [126, 60] which is a package supporting automata programming. This package adapts a *SWITCH*-technology for object-oriented programming and UML diagrams notation. In order to model the behaviour of the Diagnoser Coverability Graph (DCG) using this package, two diagrams are required. These diagrams are the Connectivity and Transition diagrams. The detail of these diagrams and how they can be used to represent the Diagnoser are described in the following sections.

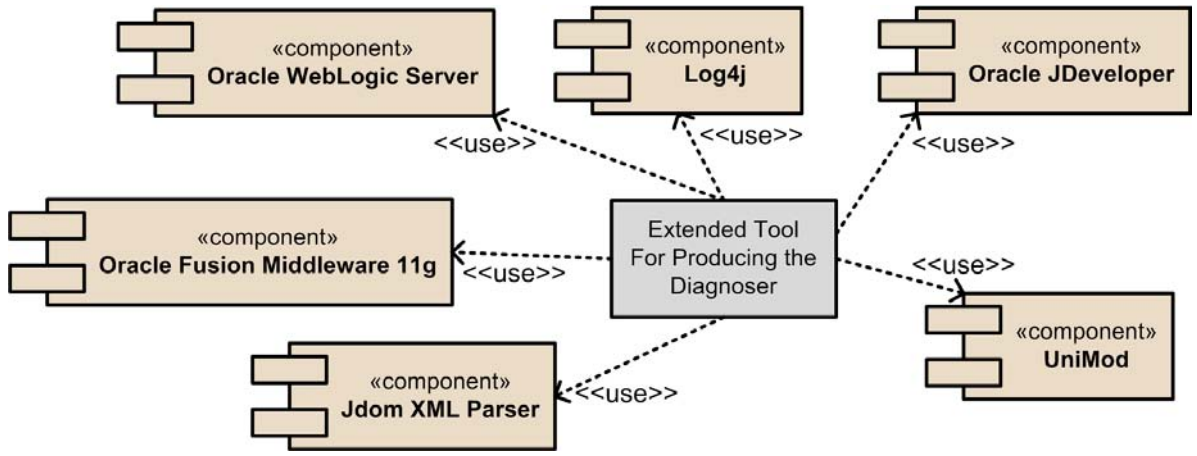

Figure 7.2: Libraries used by the extended version of our tool

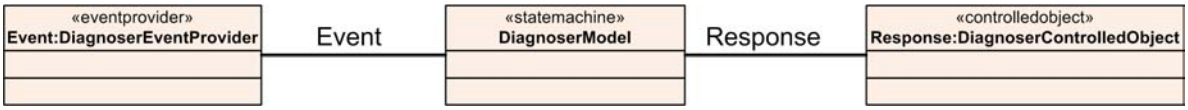

Figure 7.3: An outline of the Connectivity Diagram of the Diagnoser

7.1.1 Connectivity diagram

The Connectivity diagram is used to describe relations between a set of *finite state machines*, *event providers* and *controlled objects*. Figure 7.3 depicts an outline of the Connectivity diagram of the Diagnoser Coverability Graph (DCG). In this model, all the events of the Diagnoser are declared in an *event provider* component as a set of variables. In addition, the set of responses of the Diagnoser, which is the status of the system encrypted in the failure part, is defined in a *controlled object* component. In the following, we will show a simple example of a Connectivity diagram.

Example 7.1.1 *Figure 7.4 depicts a simple example of a Diagnoser Connectivity diagram. We assume that the Diagnoser in this example has four events and two responses. The set of the events are CheckAccount, CheckAvailability, ConfirmOrder and SendToWarehouse, while the set of responses are setNormal and setFailure1. When the system reaches a state, the responses which match the encoded information in the failure part, will be executed. The routines of these*

responses include a simple Java code to write the Diagnosing result to a Log file as follows:

```
public void setNormal(StateMachineContext context) {
     log.info("Normal");
}

public void setFailure1(StateMachineContext context) {
    log.error("Failure: one item is missing from the order.");
}
```

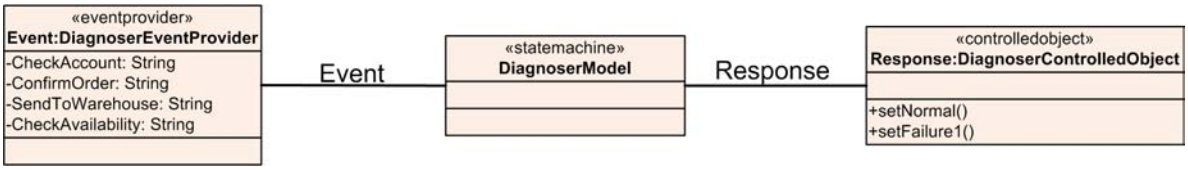

Figure 7.4: A simple example of the Connectivity Diagram

7.1.2 Transition Graph

The *Transition Graph* is a schema used to show the collaboration between the events and the states of the Diagnoser. Each state of the Diagnoser has a *name*. The Transition Graph assigns at least one response to each state. Figure 7.5 depicts a simple code snippet representing a Diagnoser state. According to the semantics of state changes presented in Section 4.4.2, each event has at least one action which is used either to add tokens to next edges or remove tokens from previous edges. For example, $addTokens(e\_x[17], 1)$ means adding one token to the edge labelled by e_x17 and $removeTokens(e\_x[6], 1)$ removes one token from the edge labelled by e_x6. Moreover, it assigns a response for each state using a method called *addAction*. The next observable nodes and the next reachable state are assigned by using *addTransition*. Figure 7.7 depicts a simple example of the *Transition Graph* of the Diagnoser of Example 7.1.1.

After implementing the Diagnoser, the tool automatically integrates the implemented Diagnoser into the system by one of the integration methods presented in Section 6.2. Finally,

```
state[3].setName("3");
state[3].addTokens(e_x[17],1);
state[3].removeTokens(e_x[6],1);
state[3].addAction(State.ON_ENTER, DiagnoserResponse.Normal);
state[3].addTransition("ConfirmOrder", state[4]);
state[3].addTransition("SendToWarehouse", state[5]);
```

Figure 7.5: Produced Code of State

the Diagnoser is deployed into the Oracle Fusion Middleware platform of the Oracle Weblogic server.

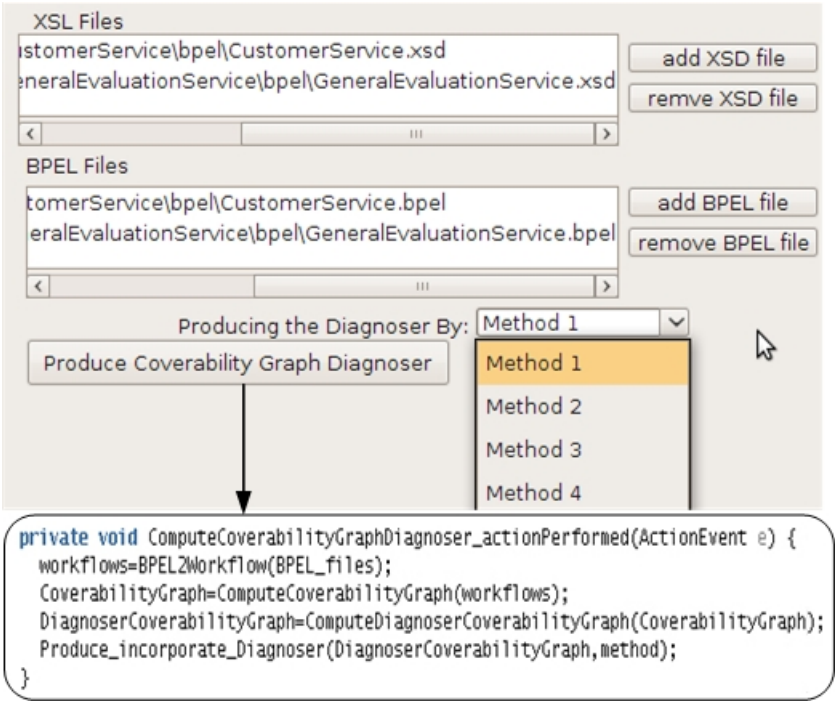

Figure 7.6: A snapshot of the implementation as an Oracle JDeveloper plugin

Our implementation has been developed as a Plugin for Oracle JDeveloper. Figure 7.6 depicts a snapshot of the tool. It only requires assigning the BPEL files, XSL files and selecting the preferred method of the integration. The tool will then carry out our approach to automatically

compute, create , implement and integrate the Diagnoser into the system.

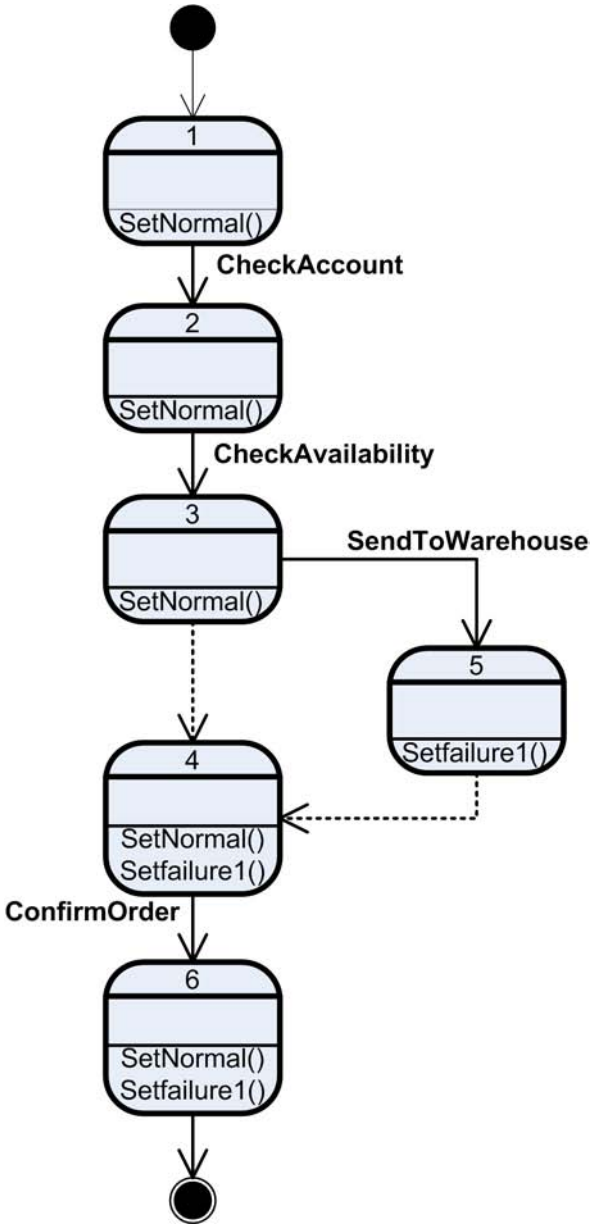

Figure 7.7: A simple example of the Transition Graph of The Diagnoser

CHAPTER 8

A CASE STUDY

In the previous chapters, we have proposed a diagnostic approach for monitoring business processes in SoA. In order to demonstrate the feasibility of our approach, we make use of a simplified case study provided by our industrial partner BT. This case study involves a common scenario called *"Resolve the Problem (RP)"* which deals with broadband issues reported by customers. Moreover, we carry out a comparison study to evaluate the integration methods presented in Chapter 6 from a performance and modularity point of view. In addition to the case studies presented in this chapter, we have carried out a study [47] in collaboration with BT based on using our approach to provide a real-time or near-real-time monitoring.

8.1 Our Approach in Practice

Using case studies is one of the most appropriate techniques for evaluating software engineering frameworks and tools. Kitchenham et al. [76, 77] propose a methodology called DESMET which could be used as a guidance for organising the evaluation study of software engineering methods through case studies. According to this methodology, the aim of the case study should be defined as a first stage [76, 77]. Therefore, the case study presented in this chapter has a simplified scenario for resolving broadband problems [9]. In this chapter, this case study will be used to demonstrate the feasibility of our approach. Moreover, we carry out a small

performance study in order to define the most suitable integration method.

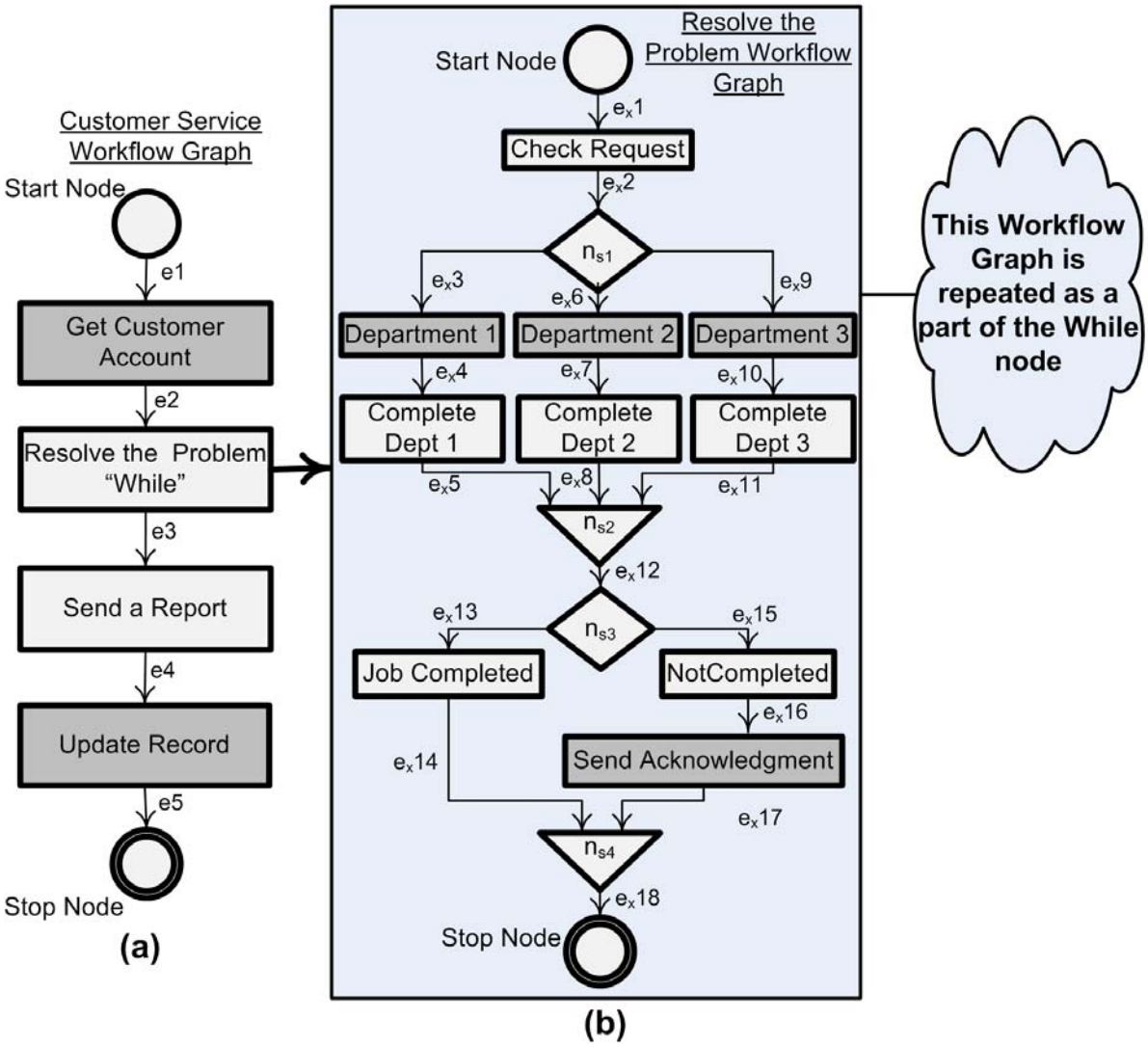

Figure 8.1: Resolve the problem Workflow Graphs

8.1.1 Case Study: Resolving Broadband Problems System

In a Telecoms company, a very common scenario is "Resolve the Problem (RP)", which deals with issues reported by customers. The BPEL model of this case study can be found in Appendix D. There are four departments involved in RP: the Customer Service department, Department 1 (in charge of back-end server related problems); Department 2 (in charge of telephone exchange related problems); and Department 3 (in charge for phone line related problems).

Suppose a customer wishes to report a broadband disconnection problem, the Customer Service would create a record for the job. As a broadband disconnection problem could be caused by various reasons, the customer service will do an initial investigation to identify the correct department for the handling of the problem. For example, a phone line related problem would be forwarded to Department 2. The selected department would subsequently carry out further investigations and resolve the problem.

However, in several situations, failures can happen. A typical failure within the Telecoms business process is related to a Right First Time (RFT) failure, which means that either a customer has been unable to resolve their issues in one request; or certain tasks of the business process have been repeated due to internal errors such as incorrect allocation of tasks. There are a number of possible reasons for a RFT failure, such as human error, incorrect results from initial investigations, runtime exceptions, software or hardware failures, and process design problems. When failure occurs, having a diagnosis method in place is considered valuable to provide the following information:

- When did the failure occur? If the diagnosis method can notify failures in real-time, the negative consequences can be occurred.
- What type of failure has occurred? A clear indication of failure type can help to speed up the failure recovery process.
- Where exactly did the failure occur? It is crucial to identify which task/service in the process has actually caused the failure. This will make the failure recovery process more focused and efficient.

8.1.2 Extracting the Extended Workflow Graphs of the running example

In order to produce a Diagnoser for the presented case study, in accordance to the approach proposed in this thesis, the following steps are carried out. Firstly, the Extended Workflow

Graph is extracted from the BPEL files. To achieve this, the set of the BPEL files of the case study are assigned to our tool which is described in Chapter 7. A Workflow Graph with Invocation and While nodes (WFGIW) is subsequently produced for each business process. The set of Workflow Graphs with Invocation and While nodes constructs the Extended Workflow Graph tree $T = (\mathscr{V}, \Sigma)$, where $\mathscr{V} = \{G_1, G_2\}$ such that G_1 and G_2 are the Workflow Graphs corresponding to the Customer Service Workflow Graph and the Workflow Graph associated with the *While* node of the Customer Service. The Customer Service Workflow Graph is a graph of form $G_1 = (N, E)$, where N =*{Start Node, Get Customer Account, Resolve the Problem (While), Send Report, Update Record, Stop Node}*, and $E = \{e_1, e_2, e_3, e_4\}$. The Customer Service Workflow Graph is depicted in Figure 8.1a. The repetitive behaviour associated to the *While* node involved in the Customer Service Workflow Graph is also a graph of form $G_2 = (N, E)$, where N =*{Start Node, Check Request,* n_{s_1} *(decision), Department 1, Department 2, Department 3, Complete Dept 1, Complete Dept 2, Complete Dept 3,* n_{s_2} *(merge),* n_{s_3} *(decision), Send Acknowledgement, NotComplete, Job Completed,* n_{s_4} *(merge), Stop Node}*, and $E = \{e_{x_1}, e_{x_2}, \ldots, e_{x_{18}}\}$. The repetitive behaviour associated to the *While* node is captured in another Workflow Graph depicted in Figure 8.1b.

State of the running Example

According to the *state* definition explained in Section 4.4.1, the state of the running example is divided into two parts, as we have two Workflow Graphs, each of which captures the movement of tokens in the Workflow Graph which corresponds to that part. These parts can be explained as follows:

The Customer Service Workflow Graph Part:

$$\underbrace{\overset{e_1 e_2 e_3 e_4 e_5 s_1^W}{1\,0\,0\,0\,0\,0}}_{\text{Customer Service},|s|=6}$$

This part, which belongs to the Customer Service Workflow Graph, has 6 coordinates, where 5

coordinates are the total number of the Workflow Graph edges (i.e. $|E|$ of G_1) and 1 coordinate is used to mark the *While* node during its execution.

While Part:

$$\underbrace{\overset{e_1}{0}\overset{e_2}{0}\overset{e_3}{0}\overset{e_4}{0}\overset{e_5}{0}\overset{e_6}{0}\overset{e_7}{0}\ldots\overset{e_{18}}{0}}_{\text{While node},|s|=18}$$

This part, which has 18 coordinates, belongs to the internal behaviour associated to the While node of the Customer Service Workflow Graph of Figure 8.1a.

As a result, the state of the Extended Workflow Graph of the given example is expressed as a combination of these two parts. Figure 8.2 depicts the initial state of the system which has only one token in the output edge of the Start node of the Customer Service Workflow Graph. This is indicated by assigning "1" to the first coordinate of the Customer Service Workflow Graph.

$$s = (\underbrace{\overbrace{100000}^{|E|=6}}_{\text{Customer Service}} \; | \underbrace{\overbrace{000000000000000000}^{|E|=18}}_{\text{Resolve the Problem Service}})$$

Figure 8.2: Initial State of the Case Study

8.1.3 Observability of the running example

As explained in Section 5.1, for each Workflow Graph with Invocation and While nodes $G = (N, E)$ of an Extended Workflow Graph (EWFG) $T = (\mathscr{V}, \Sigma)$, the set of the nodes N are classified and partitioned into disjoint subsets of observable N_{obs}, whose their occurrence can be observed; and unobservable N_{uo} representing nodes that should be unobservable for other services. The set of failure actions, which should be detected, are denoted by N_f. As explained in Section 3.2, BPEL activities are annotated with new information to represent the Observability. Therefore, such information are mapped to the Workflow Graph nodes, which are extracted from the BPEL models. These can be explained in the following manner:

- For the Customer Service Workflow Graph depicted in Figure 8.1a:

the set of observable nodes is $N_{1_{obs}}$={*Resolve the Problem (While), Send Report*}; and the set of unobservable nodes is $N_{2_{uo}}$={*Start Node, Get Customer Account, Update Record, Stop Node*}

- For the Workflow Graph associated with the While loop of the Customer Service depicted in Figure 8.1b:
  $N_{2_{obs}}$={*Department 1, Department 2, Department 3, Send Acknowledgement*}, $N_{2_{uo}}$={*Start Node, Check Request,* n_{s_1} *(decision), Complete Dept 1, Complete Dept 2, Complete Dept 3,* n_{s_2} *(merge),* n_{s_3} *(decision), NotComplete,* n_{s_4} *(merge), Stop Node*}, $N_{2_{f_1}}$={*NotComplete*}

8.1.4 The Diagnoser of the running Example

According to our approach presented in this thesis, a Diagnoser Service for the case study has computed by carrying out the following two steps. Firstly, Algorithm 2, which is presented in Section 5.3, was applied to the Extended Workflow Graph of the system; and a Coverability Graph was produced as depicted in Figure 8.3. Secondly, the Diagnoser Coverability Graph (DCG) was computed by applying Algorithm 3, which is presented in Section 5.4, to the produced Coverability Graph. The produced Diagnoser Coverability Graph is depicted in Figure 8.5.

As explained in Section 5.4.1, the Diagnoser state is similar to the state of the Coverability Graph, but the Diagnoser state has an additional part for failures. This part has a number of coordinates which is used to encode the information about failures. In our case study, the failure part has only one coordinate which is N_{f_1}, as we are dealing with only one failure which is Right-First Time failure. This failure means that either a customer cannot have their issues resolved in one request or certain tasks of the business process have been repeated due to internal errors such as incorrect allocation of tasks.

Figure 8.4 depicts a Diagnoser state which is marked with a bold border in Figure 8.5. In this state, there is only one token in the last coordinates of the Customer Service part. This

explains that the internal behaviour of the *While* loop is executing. In addition, the failure part is marked with "1" which means that at this state a failure of type "1" (i.e. Right-First Time failure) has occurred.

The produced Diagnoser can be implemented and integrated in SoA. In particular, our tool supports the four integration methods, which are explained in Chapter 6. In the following section, we will conduct a comparative study in terms of performance and modularity for these methods.

8.2 Methods comparison

8.2.1 Performance

Evaluating the resources required to implement the Diagnoser is considered a requisite task. Therefore, the four methods of integrating the Diagnoser into the system, presented in Chapter 6, have been evaluated in terms of performance. A common practise in evaluating services of SoA is to utilise the *Stress Test*. The *Stress Test* is a technique used to identify and verify the stability, capacity and the robustness of services [73, 42]. The Stress Test requires defining the number of the concurrent threads that should be allocated to the Diagnoser service, the number of loops, and the delay between invocations. With the performance statistics, we can identify any possible bottlenecks and optimise performance.

To achieve this, the Diagnoser, which is produced in Section 8.1.4, has been implemented in the four integration methods presented in Chapter 6. Then, the Stress Test has been applied to each integration method by handling a different number of concurrent threads. This is specified as 5, 10, 15, 20, 25, 30, 35, 40, 45 and 50 threads. The delay between invocations is assigned to one second. To be more accurate, for each method, the test is repeated five times. The mean of the executions time has subsequently been calculated.

The machine which is used in this test has the following configuration: Sony SZSVP, Intel Core Due 2.16GHz processor, 3G RAM.

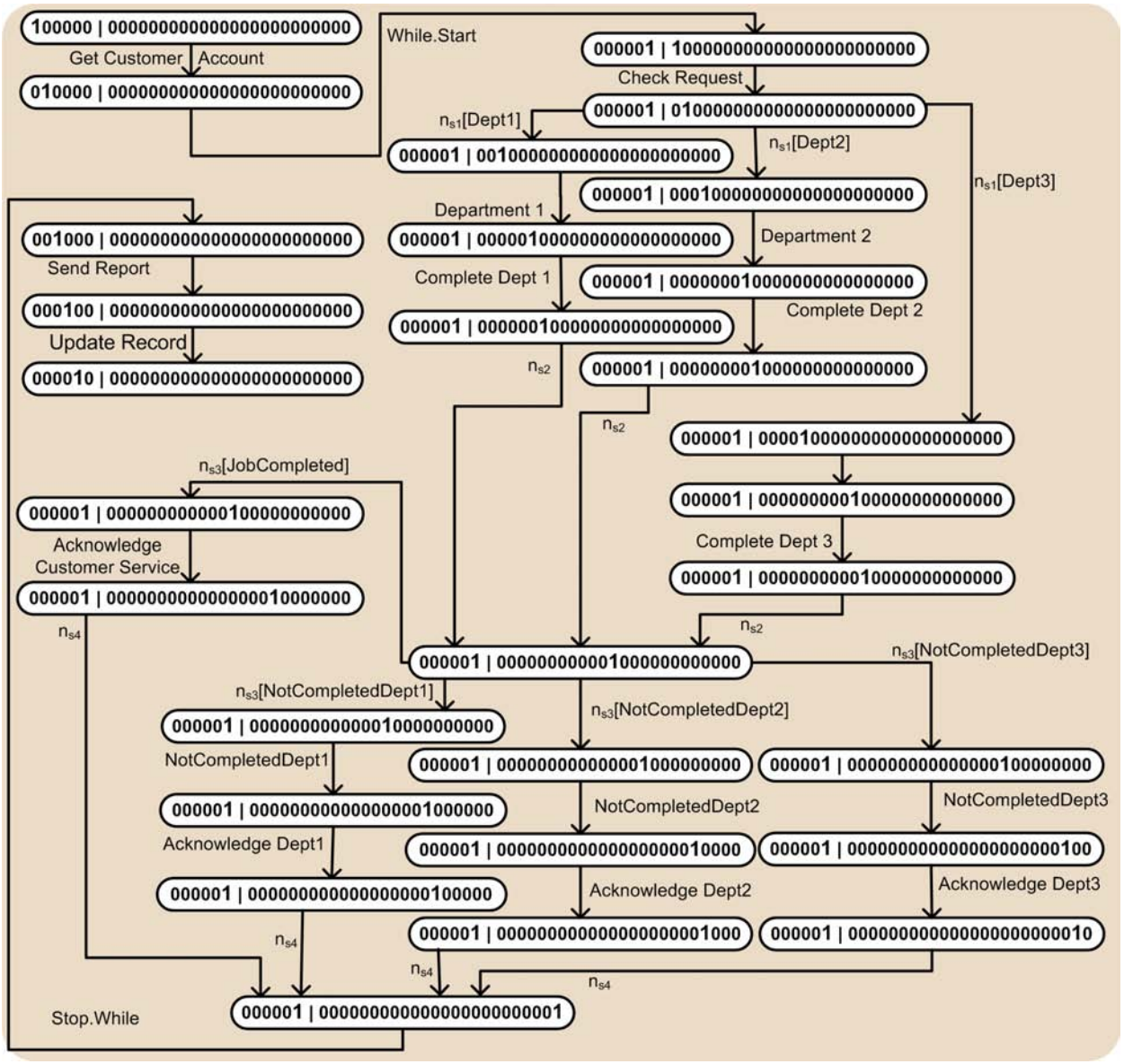

Figure 8.3: The Coverability Graph of the running example

$$s = (\ \overbrace{000001}^{|s^E|=6}\ |\overbrace{000000000000000010}^{|s^E|=18}|\overbrace{1}^{|N_{f_\ell}|=1}\ )$$

Customer Service — Resolve the Problem Service — Failures

Figure 8.4: The Diagnoser state

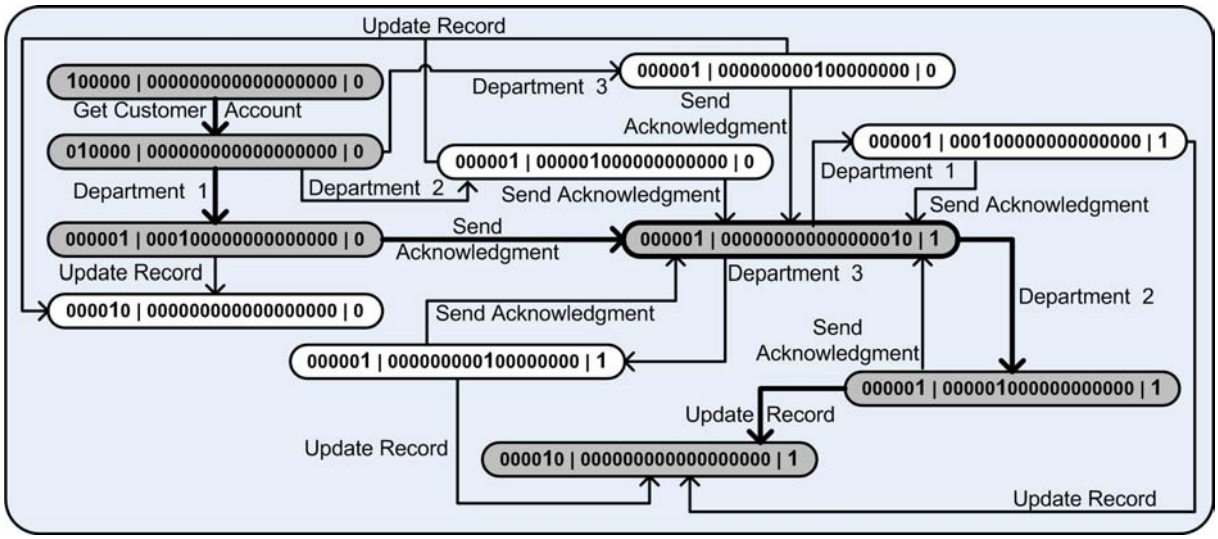

Figure 8.5: The Diagnoser Coverability Graphs of the running example

The results are depicted as a line chart in Figure 8.6 which represents the execution time (in seconds) of a number of concurrent threads. The performance of these methods can be seen as linear and parallel, where Method 3, which is based on implementing the Diagnoser as a Web service and using extra Invocations to execute the Diagnoser, shows slightly better performance as it completes processing of five threads by 0.387 seconds. Whereas, Method 2, which is based on implementing the Diagnoser as a BPEL service and using the Protocol Service to interact with the Diagnoser, accomplishes the same number withing 0.637 seconds.

It is noteworthy that the difference in execution time between these two methods was around 0.25 seconds, which is considered negligible. For more information relating to the numerical values of the Stress Test, please refer to Appendix C.

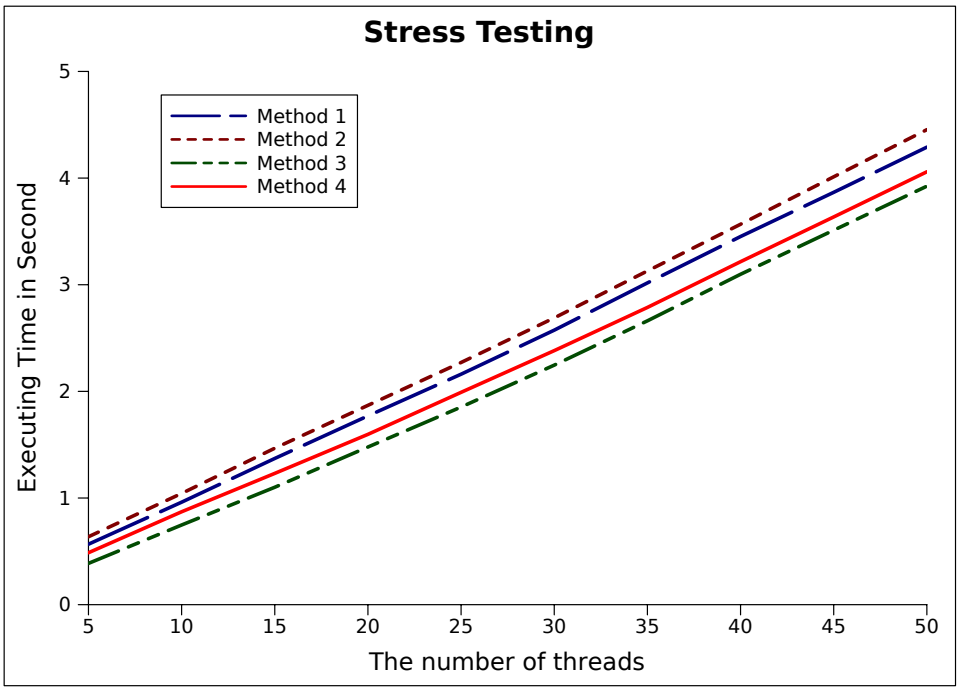

Figure 8.6: Stress Testing Result

8.2.2 Modularity

Modularity can play a key role in the early design stages of the software architecture discipline [122]. Therefore, Method 4 has greater advantages from a programming and performance point of view. This is because it is based on the use of the *Protocol Service* which results in a modularised design, which brings to the system the following benefits:

1. Reliability: using the *Protocol Service* provides faster and more reliable processes.

2. Faster and easier development. The focus would be on the functionality of code modules rather than on the mechanics of implementation.

3. Faster and easier testing.

4. Maintainability: this also makes modification of programs easier. For example, modifying the name of a system service using one of the integration methods based on utilising

the *Protocol Service*, only requires the changing of the service name in the *Protocol Service*. However, integration methods which are based on adding extra Invoke activity to execute the Diagnoser, require modification of all services that involve that service.

Moreover, Method 4 is considered as an Orchestration architecture which is a more flexible paradigm offering the following advantages over the Choreography [73]: i) the coordination of component processes is centrally managed by a known coordinator; ii) Web services can be incorporated without being aware that they are taking part in a business process; iii) alternative scenarios can be put in place in case of a fault. However, it could be argued that using the *Protocol Service* may result in bottlenecks affecting the performance of the system. Various distributed diagnosing schemes are proposed to enhance and to address this issue [132, 41, 5], which will be used as future research.

8.3 Evaluation of our Approach

8.3.1 Real-time business process diagnosis

Due to fierce market competition, an enterprise needs to closely monitor their business processes, improve services to satisfy new market needs, and quickly identify and recover any process failures. The current process monitoring technologies are mostly based on system log, which means that when a failure is identified, it has already happened. To prevent the delivery of wrong or faulty services to customers, failure should be ideally identified and recovered before the end of an execution of process. In this case, a real-time or near real-time process monitoring technique is considered essential. Almost all the current BPEL execution engines have real-time BPEL execution monitoring functionality. However, these monitoring methods are mainly provided to BEPL developers for debugging purposes.

As the Diagnoser monitors the state of a BPEL process and identifies failure in real-time, it can provide valuable information for failure recovery at runtime before the process delivers the

wrong products or services to customers. The type of the failure can also be identified at the same time so that the failure recovery process can be more effective and efficient.

8.3.2 Failure type indication

The diagnosis result obtained by the presented approach informs not only the occurrence of the failure, but also the type of the failure. The statistic function can clearly show the frequency of each type of failure over all process instances. The result can be represented as dashboard, such as Oracle, so that the Process Manager can have visibility to the status of processes execution. For example, in our simulated business process data set of the case study presented in this chapter, we have 50 process instances which are generated randomly. The occurrence percentage of Right-First time failure is 3.5%. If the Process Designer can figure out the reason for the failure, then strategic solutions can be put into place to prevent the same type of failure occurring in the future or reduce its frequency. This can be achieved through additional staff training, software or hardware upgrade, and process re-design.

8.3.3 Location of failure occurrence

Each failure identified by the proposed process diagnosis method represents a series of events that lead to the failure. Therefore, as soon as the type of a failure is confirmed, the method can discover which task/service in the process has caused the failure according to the series of events. The associated benefits can be: 1) it shortens the time of failure diagnosis; 2) the process can carry on once the recovery from failure has been achieved, rather than restarting from the beginning.

CHAPTER 9

CONCLUSION AND FUTURE WORK

This thesis has presented a method for the automated creation of the Diagnoser, which is used to monitor the behaviour of business processes in SoA. The presented method has built upon well-established algorithms proposed for diagnosing failures in Discrete Event System (DES). We have adopted and extended the conventional Workflow Graph [128] which closely follows the BPEL standards as a modelling language. Moreover, the adopted Workflow Graph was extended to include new constructs such as While and Invocation nodes which are neither supported by the standards nor adopted by tool vendors. Such nodes enhance the Workflow Graph to avoid some unsupported scenarios such as *unstructured loops* which are caused by linking the flow back to an earlier activity. Our extension is called the Extended Workflow Graph.

Building on the formalism underlying Workflow Graphs, which itself has its roots in Petri nets, we have formalised our modelling language. This has been used to prove that models created from the widely used subset of BPEL produce regular languages. This result, to the best of our knowledge, is original. This is also an important result which means the Diagnosability theory of DES can be applied in this context in order to produce the Diagnoser; as applying the Diagnosability theory of DES requires regular models.

To demonstrate that the presented formalism is suitable for the model-based diagnosis, a method of generation of Diagnosers based on Discrete Event System (DES) has been presented.

We have extended exiting algorithms of DES [118, 57, 58] to the Extended Workflow Graph. As a result, two algorithms have been put forward. The first algorithm was used to compute the Coverability Graph of the Extended Workflow Graph of the system. The second algorithm was applied to the generated Coverability Graph in order to produce the Diagnoser.

Moreover, we have harnessed the capability of the Model Driven Architecture (MDA) to propose various ways to implement and integrate the Diagnoser into the system. The first choice being to implement the Diagnoser as a BPEL service; whilst the second to implement the Diagnoser as a dedicated Java Class deployed as a Web service. To integrate the implemented Diagnoser into the system, there are also two options. An option is based on modifying each business process to include an extra *Invoke* activity to execute the Diagnoser Service after each invocation. The other option is to produce a new service called a *Protocol Service*, which is responsible for accomplishing the interaction between the Diagnoser Service and the existing services of the system. From the composing Web services prospective, we showed that using the *Protocol Service* was a more flexible method. It also provided better performance from a maintainability and modularity point of view.

The presented approach has been implemented as a plugin for Oracle JDeveloper and the tool has been built on the SiTra [6] model transformation engine. The feasibility of the approach has been demonstrated with the help of a case study offered by our industrial partner BT.

9.1 Summary of Contributions

In particular, this thesis has provided the following contributions:

- A modelling approach to the enhancement of the modelling language for specifying models of business processes. This is based on adapting and extending the conventional Workflow Graph, as explained in Chapter 4. Our extension is called the Extended Workflow Graph.

- A model-based approach based on extending the DES techniques to the proposed Ex-

tended Workflow Graph, as explained in Chapter 5.

- Proving that the Extended Workflow Graph models extracted from the widely used subset of BPEL models produce regular languages, as explained in Section 5.3.1.
- Various methods to implement and integrate the produced Diagnoser into the system, as explained in Chapter 6.
- Implementing the proposed approach as a Plugin for Oracle JDeveloper to facilitate the fully automated creation of the Diagnoser, as explained in Chapter 7.
- Demonstrating the feasibility of the presented approach with the help of a case study provided by a Telecoms company, as presented in Chapter 8. Analysis of the performance of the integration methods is carried out in order to define the best method which gives better performance.

9.2 Future Work

Following the advances made in this thesis, a number of directions for future research have arisen. Some of these extensions would help to overcome some of the limitations of this research, whilst others would provide additional capabilities.

An important next step in extending our approach consists of finding a method that will enable the user to define the system failures that the Diagnoser should detect. As explained in this thesis, we do not focus on how to model failures as it is a topic of its own. In this approach, failures have been defined manually in a Java code with the help of the annotations described in Chapter 3. In order to add more flexibility for the user, we will investigate and explore the possibility of adopting Yen's path logic [17, 138] to our approach for defining custom failures for the Diagnoser. Yen's Path Logic [17, 138] is a class of formulas for paths in Petri nets used to determine whether there exists a path in a given Petri net satisfying a given formula. If this exploring shows a positive results, we will integrate the Yen's Path Logic with our approach in

order to declare the undesirable scenarios as formulas. Then, these formulas and the Extended Workflow Graph (EWFG) of the system $T = (\mathscr{V}, \Sigma)$ will be used to produce a new Extended Workflow Graph (EWFG) $T' = (\mathscr{V}', \Sigma')$ where the undesirable scenarios is modelled as a part of the model with unobservable actions, as shown in Figure 9.1.

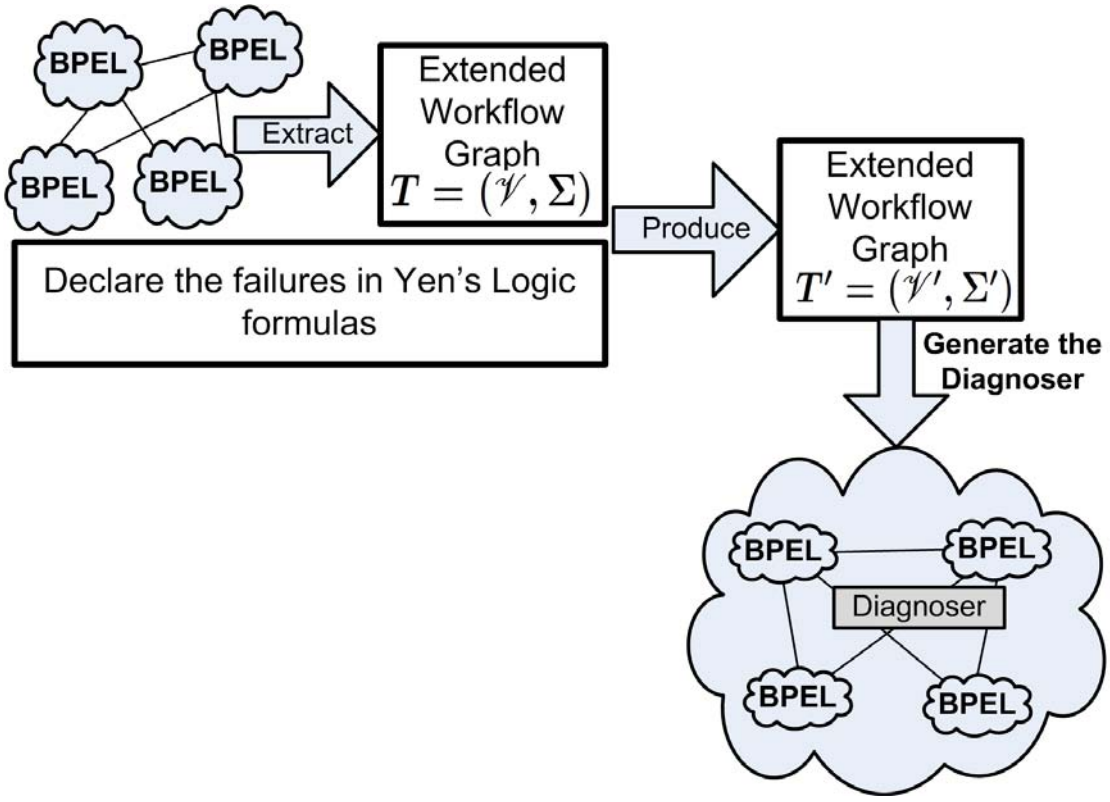

Figure 9.1: Using Yen's logic path to defining failures

Moreover, we shall apply the presented approach to a large enterprise case study which must be carefully chosen. We will make use of this case study to carry out an appropriate evaluation for our method from performance, capability, scalability and applicability point of views. In addition, this will be used to study the complexity of the Diagnoser model and the Diagnoser state.

Moreover, we shall extend the presented approach to a decentralised architecture. The decentralised diagnosis of discrete event systems has received considerable attention in dealing with large systems [133, 54, 41]. Such systems can be diagnosed by a centralised Diagnoser

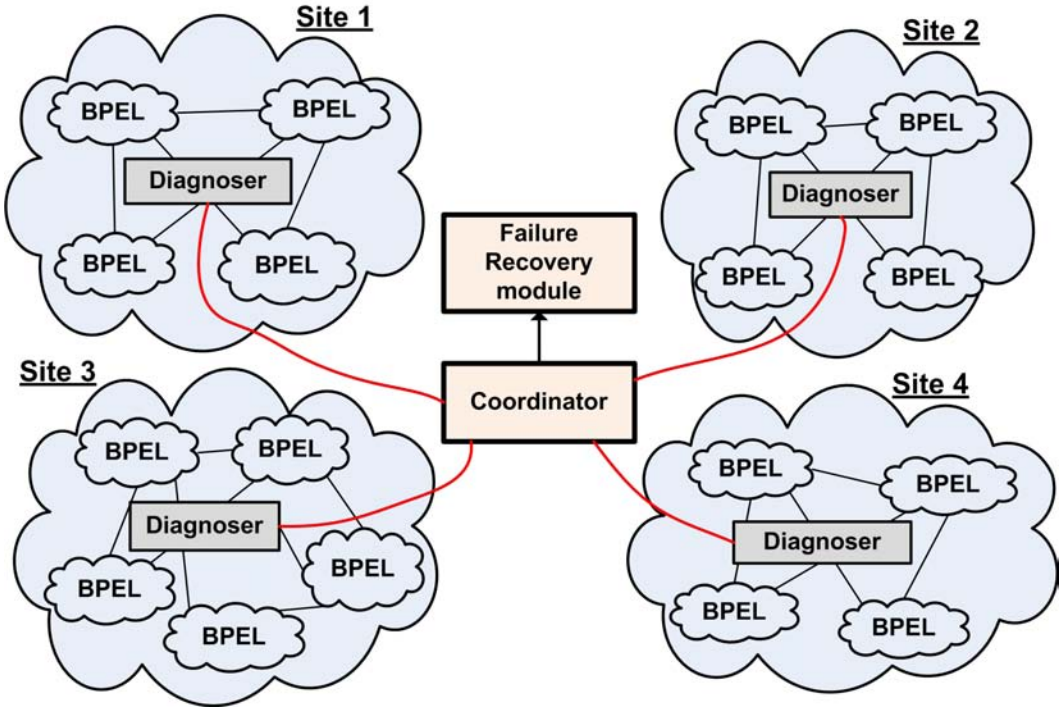

Figure 9.2: Decentralised Diagnosing Architecture

as explained in this thesis, but may result in bottlenecks affecting the performance. The decentralised diagnosis is based on decomposing the system into several local "sites". Then, a Diagnoser for each site of the system is generated as shown in Figure 9.2. Each local Diagnoser derives the local observations and performs model-based diagnosis on the basis of the projection of the system model on the local observable events. Local Diagnosers are linked with each other by a communication channel called *Coordinator* that allows exchange of the diagnostic information. Therefore, extending the method presented in this thesis along with a decentralised diagnosing approach may result in better performance and potentially reduce the state space exponentially.

APPENDIX A
BPEL LANGUAGE

The BPEL architectural model is built from various constructs. The most important BPEL constructs are those related to the invocation of web services. BPEL provides such constructs to invoke operations of web services either synchronously or asynchronously. In addition, the execution can be run in sequence or in parallel. Listed below are the most important activities that BPEL provides [73, 53]:

- **Invoke**: it is used to execute an operation of another service. This execution can be carried out in two ways which are synchronous request/response or an asynchronous one-way operation [22]. In general, executing a Web service requires using XML SOAP (Simple object Access Protocol) for executing operations, and XML WSDL(Web services Description language) for describing component interface [73].
- **Waiting** for the client to invoke the business process through sending a message, using Receive (receiving a request).
- **Reply** to return the response for synchronous BPEL process.
- **Assign** is used to manipulate data variables.
- **Throw** to indicate faults and exceptions.
- **Terminate** to terminate the entire process.

By combining these and other basic activities, more complex activities and business processes can be defined in an algorithmic manner. Such activities are described as follows [22]:

- **Sequence** is used to define a set of activities that will be invoked in an ordered sequence.
- **Flow** for capturing parallel processes.
- **Switch** is an activity that supports conditional selection based on conditions defined in *Case* elements, followed by an optional *otherwise*.
- **While** is an activity which supports the repeated execution of nesting activities as long as a *while condition* holds true.

For further information about Web services and BPEL, please refer to [11, 73]

APPENDIX B
A FRAGMENT OF THE MODEL TRANSFORMATION CODES

In this Section we present some experimental code for transforming BPEL activities to Deterministic Automaton.

B.1 Generating the Diagnoser Code in UniMod model

The following Code fragment depicts the generated code of the Diagnoser Coverability Graph of the case study discussed in Chapter 8.

```
private StateMachineConfig lookForTransition(Event event,
                StateMachineContext context, StateMachinePath path,
                StateMachineConfig config) throws Exception
  {
        BitSet calculatedInputActions = new BitSet(0);
        int s = decodeState(config.getActiveState());
        int e = decodeEvent(event.getName());
        while (true)
          {
            switch (s)
             {
                if (case.equal(GetCustomerAccount))
                   fireTransitionCandidate(context, path, "1",
                        event, "1#2#GetCustomerAccount#true");
                   fireTransitionFound(context, path, "1", event,
                     "1#2#GetCustomerAccount#true");
                   fireComeToState(context, path, "2");
                   fireBeforeOutputActionExecution(context, path,
                     "1#2#GetCustomerAccount#true", "o1.setNormal");
                   o1.setNormal(context);
                   fireAfterOutputActionExecution(context, path,
                     "1#2#GetCustomerAccount#true", "o1.setNormal");
```

```
                return new StateMachineConfig("2");
                default:
                return config;
               }
             else if (case.equal(Department1))
              {
                case Department1:
                fireTransitionCandidate(context, path, "2",
                     event, "2#3#Department1#true");
                fireTransitionFound(context, path, "2",
                             event, "2#3#Department1#true");
                fireComeToState(context, path, "3");
                fireBeforeOutputActionExecution(context,
                  path, "2#3#Department1#true", "o1.setNormal");
                o1.setNormal(context);
                fireAfterOutputActionExecution(context, path,
                   "2#3#Department1#true", "o1.setNormal");
                return new StateMachineConfig("3");
                default:
                return config;
               }
             //the rest of the Diagnoser event is added here.
          }
```

APPENDIX C
STRESS TEST RESULTS

In our study, the Stress Test has been applied to each integration method presented in Chapter 6, by handling a different number of concurrent threads. This is specified as 5, 10, 15, 20, 25, 30, 35, 40, 45 and 50 threads. To be accurate, for each method the test is repeated five times for each number of threads, and then the mean of the execution time has been calculated. In this section, we shall show the details of these results.

C.1 Method 1

Figure C.1 presents the result of the Stress Test of Method 1, which is based on implementing the Diagnoser as a BPEL service and is integrated with the help of using an extra invocation activity to execute the Diagnoser, as explained in Chapter 6.

Threads	5	10	15	20	25	30	35	40	45	50
t(s)	0.56	0.954	1.364	1.7	2.145	2.558	3	3.44	3.8545	4.278
t(s)	0.55	0.944	1.354	1.779	2.224	2.637	3.081	3.521	3.9355	4.359
t(s)	0.585	0.979	1.389	1.793	2.238	2.651	3.095	3.535	3.9495	4.373
t(s)	0.49	0.884	1.294	1.698	2.008	2.421	2.864	3.304	3.7185	4.142
t(s)	0.65	1.044	1.454	1.88	2.19	2.603	3.045	3.46	3.8745	4.298
Mean	0.567	0.961	1.371	1.77	2.161	2.574	3.017	3.452	3.8665	4.29

Figure C.1: The result of the Stress Test of Method 1

C.2 Method 2

Figure C.2 illustrates the result of the Stress Test of Method 2, which is based on implementing the Diagnoser as a BPEL service, and is integrated with the help of the *Protocol Service*, as explained in Chapter 6.

Threads	5	10	15	20	25	30	35	40	45	50
t(s)	0.695	1.043	1.47	1.85	2.299	2.66	3.131	3.554	4.011	4.446
t(s)	0.665	1.013	1.44	1.865	2.27	2.7	3.13	3.561	4.016	4.47
t(s)	0.585	1.075	1.502	1.872	2.27	2.693	3.13	3.55	4.025	4.38
t(s)	0.55	1.039	1.468	1.899	2.236	2.694	3.123	3.57	3.99	4.444
t(s)	0.69	1.045	1.455	1.859	2.28	2.698	3.131	3.6	4.018	4.53
Mean	0.637	1.043	1.467	1.869	2.271	2.689	3.129	3.567	4.012	4.454

Figure C.2: The result of the Stress Test of Method 2

C.3 Method 3

Figure C.3 shows the result of the Stress Test of Method 1, which is based on implementing the Diagnoser as a dedicated Java Class deployed as Web Service with the help of an extra Invocation activity to execute the Diagnoser, as explained in Chapter 6.

Threads	5	10	15	20	25	30	35	40	45	50
t(s)	0.379	0.739	1.099	1.48	1.85	2.19	2.647	3.099	3.5218	3.929
t(s)	0.381	0.743	1.101	1.4756	1.823	2.258	2.691	3.101	3.50144	3.933
t(s)	0.39	0.758	1.106	1.479	1.89	2.282	2.592	3.104	3.5255	3.895
t(s)	0.398	0.766	1.097	1.4787	1.831	2.2	2.678	3.089	3.5202	3.95
t(s)	0.387	0.729	1.102	1.4777	1.871	2.3	2.699	3.097	3.4871	3.913
Mean	0.387	0.747	1.101	1.4782	1.853	2.246	2.6614	3.098	3.51121	3.924

Figure C.3: The result of the Stress Test of Method 3

C.4 Method 4

Figure C.4 illustrates the result of the Stress Test of Method 2, based on implementing the Diagnoser as a dedicated Java Class deployed as Web Service; and is integrated with the help of the *Protocol Service*, as explained in Chapter 6.

Threads	5	10	15	20	25	30	35	40	45	50
t(s)	0.488	0.89	1.238	1.596	1.988	2.382	2.791	3.219	3.633	4.05
t(s)	0.485	0.872	1.232	1.589	1.989	2.394	2.806	3.218	3.641	4.07
t(s)	0.494	0.871	1.211	1.599	1.998	2.401	2.787	3.225	3.64	4.101
t(s)	0.489	0.842	1.241	1.621	2.001	2.361	2.79	3.21	3.635	4.025
t(s)	0.479	0.88	1.233	1.58	1.979	2.372	2.761	3.208	3.629	4.049
Mean	0.487	0.871	1.231	1.597	1.991	2.382	2.787	3.216	3.6356	4.059

Figure C.4: The result of the Stress Test of Method 4

APPENDIX D
BPEL MODEL OF THE CASE STUDY OF SECTION 8.1.1

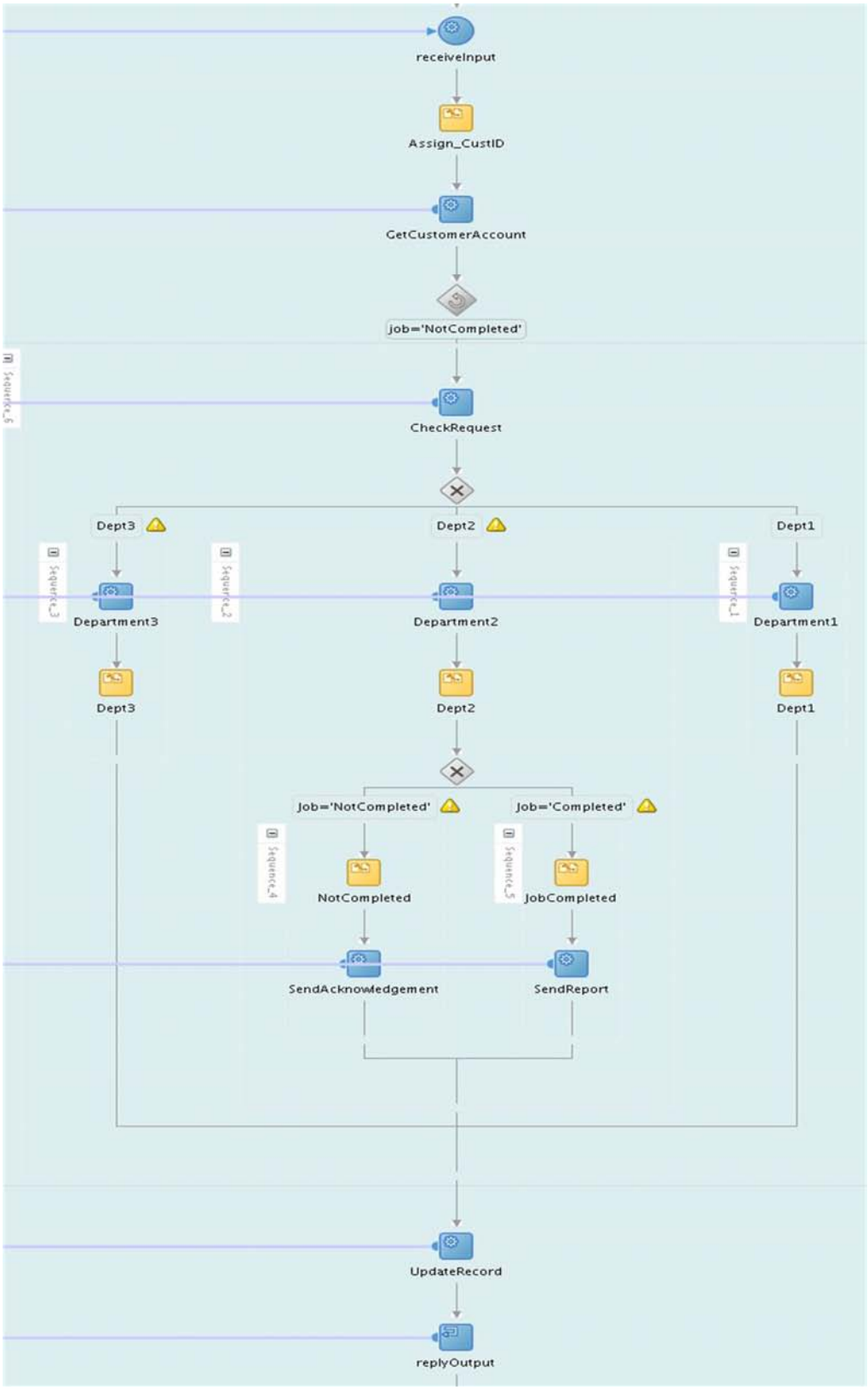

Figure D.1: BPEL model of the Case Study of section 8.1.1

LIST OF REFERENCES

[1] Jdom and xml parsing, part 1. http://www.jdom.org/downloads/docs.html.

[2] Xpdl 2.1 complete specification. Technical report, 2008.

[3] Business process modeling notation (bpmn) version 1.2. Technical report, 2009.

[4] Wil M. P. van der Aalst, Alexander Hirnschall, and H. M. W. (Eric) Verbeek. An alternative way to analyze workflow graphs. In *CAiSE '02: Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, pages 535–552, London, UK, 2002. Springer-Verlag.

[5] Armen Aghasaryan, Eric Fabre, Albert Benveniste, Renée Boubour, and Claude Jard. Fault detection and diagnosis in distributed systems: An approach by partially stochastic petri nets. *Discrete Event Dynamic Systems*, 8(2):203–231, 1998.

[6] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier. Sitra: Simple transformations in java. In *MoDELS*, volume 4199 of *LNCS*, pages 351–364, 2006.

[7] Thomas Allweyer. *BPMN 2.0*. BoD, 2010.

[8] Mohammed Alodib and Behzad Bordbar. A model driven architecture approach to fault tolerance in service oriented architectures, a performance study. In *Proceedings of the 12th Enterprise Distributed Object Computing Conference Workshops*, pages 293–300, 2008.

[9] Mohammed Alodib, Behzad Bordbar, Xiaofeng Du, and Basim Majeed. On diagnosis of failures in business processes involving iterations. In *Submitted for publication*.

[10] Mohammed Alodib, Behzad Bordbar, and Basim Majeed. A model driven approach to the design and implementing of fault tolerant service oriented architectures. In *3rd International Conference on Digital Information Management (ICDIM)*, pages 464–469, 2008.

[11] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer Berlin, 2004.

[12] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services Concepts, Architectures and Applications*. Springer, 2004.

[13] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. Softw. Eng.*, 18(3):237–251, 1992.

[14] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, 2007.

[15] Arcstyler. Arcstyler 5.0- interactive objects. www.interactive-objects.com, 2005.

[16] Liliana Ardissono, Luca Console, Anna Goy, Giovanna Petrone, Claudia Picardi, Marino Segnan, and Daniel Dupre. Cooperative model-based diagnosis of web services. In *In 16th International Workshop on Principles of Diagnosis (DX*, pages 125–132, 2005.

[17] Mohamed Faouzi Atig and Peter Habermehl. On yen's path logic for petri nets. In *Proceedings of the 3rd International Workshop on Reachability Problems*, pages 51–63. Springer-Verlag, 2009.

[18] ATLAS. Atlas, universit de nantes. http://www.eclipse.org/atl/, 2005.

[19] Siddharth Bajaj, Don Box, Dave Chappell, Francisco Curbera, Glen Daniels, Phillip Hallam-Baker, Maryann Hondo, Chris Kaler, Dave Langworthy, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, Claus von Riegen, Daniel Roth, Jeffrey Schlimmer, Chris Sharp, John Shewchuk, Asir Vedamuthu, Ümit Yalçinalp, and David Orchard. Web services policy 1.2 - framework (WS-policy). Technical report, W3C, 2006.

[20] Luciano Baresi, Sam Guinea, Olivier Nano, and George Spanoudakis. Comprehensive monitoring of bpel processes. *IEEE Internet Computing*, 14(3):50–57, 2010.

[21] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella. Diagnosis of large active systems. *Artif. Intell.*, 110:135–183, 1999.

[22] BEA, IBM, Microsoft, AG SAP, and Siebel Systems. Business process execution language for web services. version 1.1, 2003.

[23] Albert Benveniste, Eric Fabre, Stefan Haar, and Claude Jard. Diagnosis of asynchronous discrete-event systems: A net unfolding approach. *IEEE Transactions on Automatic Control*, 48:2003, 2003.

[24] T Berners-Lee. World wide web seminar,http://www.w3.org/talks/general.html, 1991.

[25] Norbert Bieberstein, Sanjay Bose, Marc Fiammante, Keith Jones, and Rawn Shah. *Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[26] S Blanvalet. *BPEL Cookbook: Best Practices for SOA-based integration and composite applications development*. PACKT PUBLISHING, 2006.

[27] Lossan Bondé, Pierre Boulet, and Jean-Luc Dekeyser. Traceability and Interoperability at Different Levels of Abstraction in Model Transformations. In *Forum on Specification and Design Languages, FDL'05*, Lausanne Suisse, 2005.

[28] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture, w3c working group note. *World Wide Web Consortium, article available from: http://www. w3. org/TR/ws-arch*, 2004.

[29] Behzad Bordbar, Gareth Howells, Michael Evans, and Athanasios Staikopoulos. Model transformation from owl-s to bpel via sitra. In *Proceedings of the 3rd European conference on Model driven architecture-foundations and applications*, ECMDA-FA'07, pages 43–58, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] Behzad Bordbar and A. Staikopoulos. On behavioural model transformation in web services. In *eCOMO*, volume 3289, China, 2004. Springer Verlag.

[31] A. Boufaied, A. Subias, and M. Combacau. Chronicle modeling by petri nets for distributed detection of process failures. In *IEEE Conference on Systems, Management and Cybernetics, Hammamet,*, pages 299–303, Tunisie, 2002.

[32] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1, 2000.

[33] Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. Identification of unbounded petri nets from their coverability graph. In *45th IEEE Conference on Decision and Control*, pages 434–440, 2006.

[34] M.P. Cabasino, A. Giua, S. Lafortune, and C. Seatzu. Diagnosability analysis of unbounded petri nets. In *CDC09: Proceedings of the 48th IEEE Conference on Decision and Control*, Shanghai, China, 2009.

[35] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. pages 341–354. Springer-Verlag, 1995.

[36] Christos Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2007.

[37] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0, 2006.

[38] Marco Comuzzi and George Spanoudakis. Dynamic set-up of monitoring infrastructures for service based systems. In *Proceedings of the ACM Symposium on Applied Computing*, pages 2414–2421. ACM, 2010.

[39] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London, 3 edition, 1972.

[40] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM System Journal*, 43:136–158, January 2004.

[41] R. Debouk, S. Lafortune, and D. Teneketzis. A coordinated decentralized protocols for failure diagnosis of discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 10(1):33–86, 2000.

[42] University of Minnesota Department: Software Verification. Stress test strategy.

[43] E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, pages 27–33, 1979.

[44] Dean Kuo Div, Dean Kuo, Michael Lawley, Chengfei Liu, and Maria Orlowska. A general model for nested transactional workflows. In *Proceedings of Workshop on Advanced Transaction Models and Architectures*, pages 18–35, 1996.

[45] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series)*. Addison Wesley, 2005.

[46] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998.

[47] Xiaofeng Du, Behzad Bordbar, Mohammed Alodib, and Basim Majeed. Applying protocol service for the monitoring of business process. In *The IEEE GCC Conference and Exhibition*, pages 633–636, 2011.

[48] Abdelkarim Erradi, Piyush Maheshwari, and Vladimir Tosic. Ws-policy based monitoring of composite web services. In *Proceedings of the Fifth European Conference on Web Services*, pages 99–108, Washington, DC, USA, 2007. IEEE Computer Society.

[49] Eric Fabre, Albert Benveniste, Stefan Haar, and Claude Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15:33–84, 2005.

[50] M. Fasbinder. Using loops in websphere business modeler v6 to improve simulations and xport to bpel. Technical report, WebSphere Software Technical Sales, IBM, 2007.

[51] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[52] Paul Fremantle, Sanjiva Weerawarana, and Rania Khalaf. Enterprise services. *Commun. ACM*, 45:77–82, 2002.

[53] Michael Friess, Erich Fussi, Dieter Konig, Gerhard Pfau, Stefan Ruttinger, Friedemann Schwenkreis, and Claudia Zentner. Websphere process server v6 - business process choreographer: Concepts and architecture. *IBM Group*, 2006.

[54] S. Genc and S Lafortune. Distributed diagnosis of place-bordered petri nets. *IEEE Transactions on Automation Science and Engineering*, 4(2):206–219, 2007.

[55] Sahika Genc and Stephane Lafortune. Distributed diagnosis of discrete-event systems using petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003),*, pages 316–336. Springer-Verlag, 2003.

[56] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases*, 3:119–153, 1995.

[57] A. Giua and C. Seatzu. Observability of place/transition nets. *IEEE Transactions on Automatic Control*, 47(9):1424–1437, 2002.

[58] A. Giua and C. Seatzu. Fault detection for discrete event systems using petri nets with unobservable transitions. In *44th IEEE Conference on Decision and Control*, pages 6323–6328, 2005.

[59] Richard Grimes and Dr Richard Grimes. *Professional Dcom Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1997.

[60] Vadim S. Gurov, Maxim A. Mazin, Andrey S. Narvsky, and Anatoly A. Shalyto. Unimod: Method and tool for development of reactive object-oriented programs with explicit states emphasis. In *Proceedings of St. Petersburg IEEE Chapters*, pages 106–110, 2005.

[61] C.N. Hadjicostis and G. Verghese. Power systems monitoring based on relay and circuit breaker. In *IEEE Procedings on Generation Transmission and Distribibution*, pages 299–303, 2000.

[62] Walter Hamscher, Luca Console, and Johan de Kleer, editors. *Readings in model-based diagnosis*. Morgan Kaufmann Publishers Inc., USA, 1992.

[63] Janelle B. Hill, Michele Cantara, Eric Deitert, and Marc Kerremans. Magic quadrant for business process management suites. Technical report, Gartner Research, 2007.

[64] William M. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, New York, NY, USA, 1982.

[65] IBM. Websphere, http://www-01.ibm.com/software/websphere/.

[66] IBM, Microsoft, and and others. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[67] Aberdeen Group Inc. Boston, ma. 2006. value of soa, computerworld, 2006.

[68] Lucas Jellema and Lonneke Dikmans. *Oracle SOA Suite 11g Handbook*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[69] Shengbing Jiang and Ratnesh Kumar. Failure diagnosis of discrete-event systems with linear-time temporal logic specifications. *IEEE Transactions on Automatic Control*, 49(6):934–945, 2004.

[70] G. Jiroveanu and R. K. Boel. A distributed approach for fault detection and diagnosis based on time petri nets. *Math. Comput. Simul.*, 70:287–313, 2006.

[71] George Jiroveanu, René K. Boel, and Behzad Bordbar. On-line monitoring of large petri net models under partial observation. *Discrete Event Dynamic Systems*, 18(3):323–354, 2008.

[72] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, 2005.

[73] Matjaz B. Juric, Benny Mathew, and Poornachandra Sarang. *Business Process Execution Language for Web Services*. Packt Publishing, 2004.

[74] Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11:57–81, March 2003.

[75] kermeta. http://www.kermeta.org/.

[76] B. Kitchenham, S. Linkman, and D. Law. Desmet: a methodology for evaluating software engineering methods and tools. *In IEEE Computing and Control Journal*, 8:120–126, 1997.

[77] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12:52–62, 1995.

[78] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture-Practice and Promise,*. Addison-Wesley, 2003.

[79] J. Koehler and J. Vanhatalo. Process anti-patterns: How to avoid the common traps of business process modeling, part 1. Technical report, IBM WebSphere Developer Technical Journal, 2007.

[80] Jana Koehler and Rainer Hauser. Untangling unstructured cyclic flows - a solution based on continuations. pages 121–138, 2004.

[81] H. Kreger. Web services conceptual architecture. IBM Software Group, 2001.

[82] Jean-Claude Laprie. Dependability - its attributes, impairments and means. *B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, editors, Pridictably Dependable Computing Systems*, pages 27–40, 1995.

[83] Ruben Lara, Holger Lausen, Sinuhe Arroyo, Jos de Bruijn, and Dieter Fensel. Semantic web services: description requirements and current technologies. In *International Workshop on Electronic Commerce, Agents, and Semantic Web Services, In conjunction with the Fifth International Conference on Electronic Commerce (ICEC 2003)*, Pittsburgh, PA, USA, 2003.

[84] James Lawler and H. Howell-Barber. *Service-Oriented Architecture: SOA Strategy, Methodology, and Technology*. Auerbach Publications, 2007.

[85] Xavier Le Guillou, Marie-Odile Cordier, Sophie Robin, and Laurence Rozé. Chronicles for On-line Diagnosis of Distributed Systems. Research report, 2008.

[86] Xavier Le Guillou, Marie-Odile Cordier, Sophie Robin, and Laurence Rozé. Chronicles for on-line diagnosis of distributed systems. In *Proceeding of ECAI*, pages 194–198, The Netherlands, 2008.

[87] Frank Leymann. Web services: Distributed applications without limits. In *Proc. BTW'03*, pages 26–28, 2003.

[88] Demed L'Her, Heidi Buelow, Jayaram Kasi, Manas Deb, and Prasen Palvankar. *Getting Started With Oracle SOA Suite 11g R1 A Hands-On Tutorial*. Pakt Publishing, 2009.

[89] Peter Linz. *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc., USA, 2006.

[90] C. Mancel, P. Lopez, N. Riviere, and R. Valette. Relationships between petri nets and constraint graphs: application to manufacturing. In *15th IFAC Triennial World Congress*, 2002.

[91] David W. McCoy and Yefim V. Natis. Service-oriented architecture: Mainstream straight ahead. Technical report, Gartner Research, 2003.

[92] Duncan Mills, Peter Koletzke, and Avrom Roy-Faderman. *Oracle JDeveloper 11g Handbook*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[93] MOF. Meta object facility (mof) 2.0 core specification, object management group, available at www.omg.org,, 2004.

[94] Jean J. Moreau, Roberto Chinnici, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Technical report, W3C, 2006.

[95] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceeding of the 17th international conference on World Wide Web*, pages 815–824, New York, NY, USA, 2008. ACM.

[96] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[97] S. Oasi. The Universal Description, Discovery and Integration (UDDI) protocol.

[98] omg. http://www.omg.org/corba/.

[99] OMG. Mda guide version 1.0.1, 2003.

[100] OMG. *MOF QVT Final Adopted Specification*, 2005. OMG doc. ptc/05-11-01.

[101] OMG. *OCL 2.0*, 2006. OMG doc. ptc/06-05-01.

[102] OptimalJ. Compuware software coporation, 2005.

[103] Oracle. http://oracle.com/fusionmiddleware11g.

[104] Oracle. http://oracle.com/middleware.

[105] Oracle. Oracle fusion middleware 11g. Technical report.

[106] oracle. www.oracle.com.

[107] Oracle. Oracle business activity monitoring, 2005.

[108] E. O'Tuathail and M. Rose. Using the simple object access protocol (soap) in blocks extensible exchange protocol (beep). Technical report, United States, 2002.

[109] C.M. Ozveren and A.S. Willsky. Observability of discrete event dynamic systems. *Transactions on Automatic Control*, 35:797–806, 1990.

[110] Michael P. Papazoglou. A survey of web service technologies, 2004.

[111] H. Petritsch. Service-oriented architecture (soa) vs. component based architecture. Technical report, Vienna University of Technology, Vienna, 2006.

[112] J. Poulin and A. Himler. The roi of soa based on traditional component reuse. Technical report, LogicLibrary, Inc.

[113] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[114] Wasim Sadiq and Maria E. Orlowska. Applying graph reduction techniques for identifying structural conflicts in process models. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, pages 195–209. Springer-Verlag, 1999.

[115] Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25(2):117–134, 2000.

[116] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad P. A. van Moorsel, and Fabio Casati. 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*, pages 28–41, London, UK, 2002. Springer-Verlag.

[117] M Sampath, S Lafortune, and D Teneketzis. Active diagnosis of discrete event systems. In *IEEE Transactions on Automatic Control*, pages 908–929, 1998.

[118] M. Sampath, R. Sengupta, and S Lafortune. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40:1555–75, Sept. 1995.

[119] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, 1996.

[120] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[121] Seyyed Shah, Kyriakos Anastasakis, and Behzad Bordbar. Using traceability for reverse instance transformations with SiTra. In *Design and Architectures for Signal and Image Processing (DASIP 2008). Special Session on Formal Models, Transformations and Architectures for Reliable Embedded System Design.*, Bruxelles, Belgium, 2008.

[122] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[123] James Skene, Davide D. Lamanna, and Wolfgang Emmerich. Precise service level agreements. *International Conference on Software Engineering*, pages 179–188, 2004.

[124] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures, 2004.

[125] Vladimir Tosic, Bernard Pagurek, Kruti Patel, Babak Esfandiari, and Wei Ma. Management applications of the web service offerings language (wsol). *Information System*, 30:564–586, 2005.

[126] UniMod. http://unimod.sourceforge.net.

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

[128] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 43–55. Springer-Verlag, 2007.

[129] w3. http://www.w3.org/.

[130] W3C. Web services choreography description language version 1.0, 2005.

[131] Yin Wang, Terence Kelly, and Stephane Lafortune. Discrete control for safe execution of it automation workflows. In *EuroSys*, pages 305–314, 2007.

[132] Yin Wang, Tae-Sic Yoo, and Stéphane Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17(2):233–263, 2007.

[133] Yin Wang, Tae-Sic Yoo, and Stephane Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17(2), 2007.

[134] Steve K. Wood, David H. Akehurst, Oleg Uzenkov, W. G. J. Howells, and Klaus D. McDonald-Maier. A model-driven development approach to mapping uml state diagrams to synthesizable vhdl. *IEEE Trans. Comput.*, 57:1357–1371, 2008.

[135] xactium. Xmf-mosaic: xactium. In *http://www.xactium.com/*, 2005.

[136] Yuhong Yan and P. Dague. Modeling and diagnosing orchestrated web service processes. In *IEEE International Conference on Web Services*, volume 9, pages 51 – 59, 2007.

[137] Yuhong Yan, Y. Pencole, M.-O. Cordier, and A. Grastien. Monitoring web service networks in a model-based approach. In *ECOWS05*, 2005.

[138] Hsu-Chun Yen. A unified approach for deciding the existence of certain petri net paths. *Information and Computation*, 96(1):119–137, 1992.

[139] S.H. Zad, R.H. Kwong, , and W.M. Wonham. Fault diagnosis in discrete-event systems: framework and model reduction. *IEEE Transactions on Automatic Control*, 48:1199–1212, 2003.