2011

MAPPING BPEL PROCESSES TO DIAGNOSTIC MODELS

Hamza Ghandorh

MAPPING BPEL PROCESSES TO DIAGNOSTIC MODELS

(Thesis format: Monograph)

by

Hamza Ghandorh

Graduate Program in Computer Science

A thesis report submitted in partial fulfillment

of the requirements for the degree of

Master of Science

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO

School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

The thesis by

Hamza Ghandorh

entitled:

Mapping BPEL Processes to Diagnostic Models

is accepted in partial fulfillment of the

requirements for the degree of

Master of Science

Tuesday, December 20, 2011
Date

....
Prof.
Chair of the Thesis Examination Board

Abstract

Web services are loosely-coupled, self-contained, and self-describing software modules that perform a predetermined task. These services can be linked together to develop an application that spans multiple organizations. This linking is referred to as a composition of web services. These compositions potentially can help businesses respond more quickly and more cost-effectively to changing market conditions. Compositions can be specified using a high-level workflow process language.

A fault or problem is a defect in a software or software component. A system is said to have a failure if the service it delivers to the user deviates from compliance with the system specification for a specified period of time. A problem causes a failure. Failures are often referred to as symptoms of a problem. A problem can occur on one component but a failure is detected on another component. This suggests a need to be able to determine a problem based on failures. This is referred to as fault diagnosis.

This thesis focuses on the design, implementation and evaluation of a diagnostic module that performs automated mapping of a high-level specification of a web services composition to a diagnostics model. A diagnosis model expresses the relationship between problems and potential symptoms. This mapping can be done by a third party service that is not part of the application resulting from the composition of the web services. Automation will allow a third party to do diagnosis for a large number of compositions and should be less error-prone.

Keywords: Web Service Composition Diagnosis, Codebook Technique, BPEL Mapping.

Acknowledgements

First of all I would like to thank Allah who gave me the strength and patience to learn and accomplish my dream of getting M.Sc. I am thankful to the Ministry of High Education of Saudi Arabia and the Saudi Cultural bureau for being a primary partner in this success. I would like to thank my supervisor, Prof. Hanan Lutfiyya, for her guidance and assistance. She was very helpful and encouraging in choosing the topic of this thesis and pursuing the research. Thanks to her for the help and inspiration she extended. I am grateful to the people from the Distributed and Grid Research Group (DiGS) for their help and constructive feedback. I would like also to thank faculty members and staff of the Department of Computer Science for making my studying here highly enjoyable. In addition, I would like to thank my grad classmates for their suggestions to complete this work. And many thanks go to my beloved parents, wife, son and friends for their understanding and endless love through the duration of my Masters program.

Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

This chapter provides the motivation for the research presented in this thesis. Section 1.1 defines web services composition. Section 1.2 states the problem statement. Section 1.3 shows how this thesis is organized.

1.1 Introduction about Web Services Compositions

Web services are loosely-coupled, self-contained, and self-describing software modules that perform a predetermined task. These services are physically distributed and are able to communicate using SOAP messages. These services can be linked together to develop an application that spans multiple organizations. This linking is referred to as a composition of web services. These compositions potentially can help businesses respond more quickly and more cost-effectively to changing market conditions. Compositions can be specified using a workflow process language e.g., Business Process Execution Language (WS-BPEL). WS-BPEL or BPEL is an XML-based block-structured language that specifies actions within compositions and its services [2]. An example of BPEL is shown in figure 1.1. Figure 1.1 shows a simple BPEL process that receives a greeting phrase, composes a greeting phrase and replies with the greeting. Because BPEL is intended for business process designers, it has to be graphically modelled to be readable by human. BPEL has a few graphical modelling standards. The

```
<process name="Hello" targetNamespace="http://jbpm.org/examples/hello"
  xmlns.tns="http://jbpm.org/examples/hello"
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
    <!-- establishes the relationship with the caller agent -->
    <partnerLink name="caller" partnerLinkType="tns:Greeter-Caller"
    myRole="Greeter" />
  </partnerLinks>

  <variables>
    <!-- holds the incoming message -->
    <variable name="request" messageType="tns:nameMessage" />
    <!-- holds the outgoing message -->
    <variable name="response" messageType="tns:greetingMessage" />
  </variables>

  <sequence name="MainSeq">

    <!-- compose a greeting phrase -->
    <assign name="ComposeGreeting">
      <copy>
        <from expression="concat('Hello,',bpel:getVariableData('request','name'),'!')" />
        <to variable="response" part="greeting" />
      </copy>
    </assign>

    <!-- send greeting back to caller -->
    <reply name="SendGreeting" operation="sayHello" partnerLink="caller"
      portType="tns:Greeter" variable="response" />

  </sequence>

</process>
```

Figure 1.1: Hello BPEL Example [1]

most commonly used standard to model BPEL processes for humans is referred to as Business Process Modeling Notation(BPMN). An example of BPMN is shown in figure 1.2 [2].

1.2 Thesis Focus

In order to guarantee the consistency of web services composition's workflow execution within its distributed environment, a vision of the whole interactions of web services is necessity. Such vision is gained based on the execution of an automated management system with diagnostics capabilities. Our work provides a diagnostic facility to the management system with all possible faulty web services interactions. The diagnostic facility will assist to provide automated self-healing capabilities, which will be a key feature in a web services industrial future.

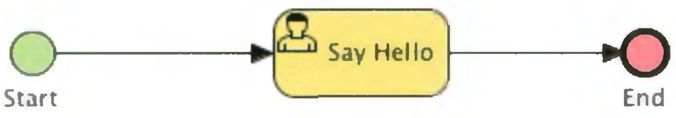


Figure 1.2: Hello BPMN Example [2]

This thesis focuses on the design, implementation and evaluation of a diagnostic module, which is referred to as diagnosis module, that performs automated mapping of a high-level specification of a web services composition to a diagnostics model. A diagnosis model expresses the relationship between problems and potential symptoms. Fault localization software can analyse instances of the diagnostics model to determine faults. This mapping can be done by a third party service that is not part of the application resulting from the composition of the web services. Automation will allow a third party to do diagnosis for a large number of compositions and should be less error-prone.

1.3 Thesis Outline

This thesis is organized as follows: Chapter 2 provides the background and related work, Chapter 3 presents the proposed approach for this research, Chapter 4 describes the architecture of the third party system, Chapter 5 describes how the diagnosis module was implemented, Chapter 6 concludes the thesis.

Chapter 2

Background

This chapter presents key definitions and concepts, and reviews the current research relevant to fault diagnosis within web services composition. In Sections 2.1 and 2.2, the basic concepts that describe compositions and underling standards are mentioned. Section 2.3 presents run-time attributes that facilitate the analysis of dynamic behaviours of services in compositions. Section 2.4 shows how these attributes can be used to form agreements between a service provider and client. Section 2.5 presents the role of management in applications composed of web services. Section 2.6 discusses the monitoring of web services. Finally, Section 2.7 presents common definitions, diagnosis process, and some related work with respect to fault diagnosis.

2.1 Web Services Compositions

This Section introduces web services compositions and some of their aspects.

2.1.1 Definition

Services can be linked together to develop an application that spans multiple organizations. This linking is called *composition*. These services are physically distributed. These services are able to communicate using messages [8]. Compositions can be conducted by a third party component. This is referred to as *orchestration*. An alternative approach is to use a *choreography* description. In the orchestration approach, the third party component, which is central, deals with a workflow of services and defines when and how compositions' members would interact. In the choreography approach, designers have a member service interact based on the previously agreed operating steps without the need for a central conductor. The choreography approach is discussed in more detail in [9].

Services can be used in multiple applications and thus are reusable. A service of a particular type can be replaced by another service if necessary. Services can be formed at anytime. Applications are flexible in that they can change topological structure, interdependencies, and workloads at run time [10].

The following is an example that illustrates a possible composition. A client wants to apply for a loan from a bank. The bank needs to obtain the credit rating for the client. Such service is provided by a credit composition, which has access to the entire credit history of individuals in a geographical zone. Therefore, the bank needs to subscribe to the credit composition. As soon as clients apply for the loans, the bank will integrate needed services from the credit composition into its own loan approval business processes.

Compositions are implemented using a workflow process language. Web Services Business Process Execution Language (WS-BPEL), which was developed by OASIS [11], is considered a standard for compositions modelling. It is an orchestration language which defines roles that take part in the message exchanges, what functions must be supported and so on.

2.2 Enabling Standards

For services accessed over the Internet several standards are needed to facilitate discovery of services and communication with the services. This Section describes several aspects of these standards.

2.2.1 XML

XML (Extensible Markup Language) is an open standard language used to describe and transmit formatted data, which was developed by W3C. It is a set of defined rules used to encode web documents in a machine-readable format [9]. It is a universal method to facilitate exchange information between Internet-based applications.

2.2.2 SOAP

SOAP (Simple Object Access Protocol) is an open standard XML-based exchange message protocol. SOAP is a simple and lightweight means for exchanging information between peers over a network [3]. It relies on RPC and HTTP for message transmission. There are two types of SOAP messages: request and response. Any SOAP message should consist of the following attributes: SOAP envelope, SOAP encoding rules, and SOAP RPC representation. SOAP request and response messages examples are presented and denoted in Figure 2.1 and Figure 2.2 [3]. In Figure 2.1, `GetLastTradePrice` function is being called with one parameter, which is a stock symbol (DIS) in <`symbol`> tag, from a service, which is StockQuote. In Figure 2.2, the service will reply with the last trade price as a numerical value in <`Price`> tag as output of the invoked function, which is denoted in Figure 2.2.

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAPENV:
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <SOAP-ENV:Body>
                <m:GetLastTradePrice xmlns:m="Some-URI">
                        <symbol>DIS</symbol>
                </m:GetLastTradePrice>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.1: SOAP request message example [3]

```
<SOAP-ENV:Envelope
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        SOAPENV:
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        <SOAP-ENV:Body>
                <m:GetLastTradePriceResponse xmlns:m="Some-URI">
                        <Price>34.5</Price>
                </m:GetLastTradePriceResponse>
        </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

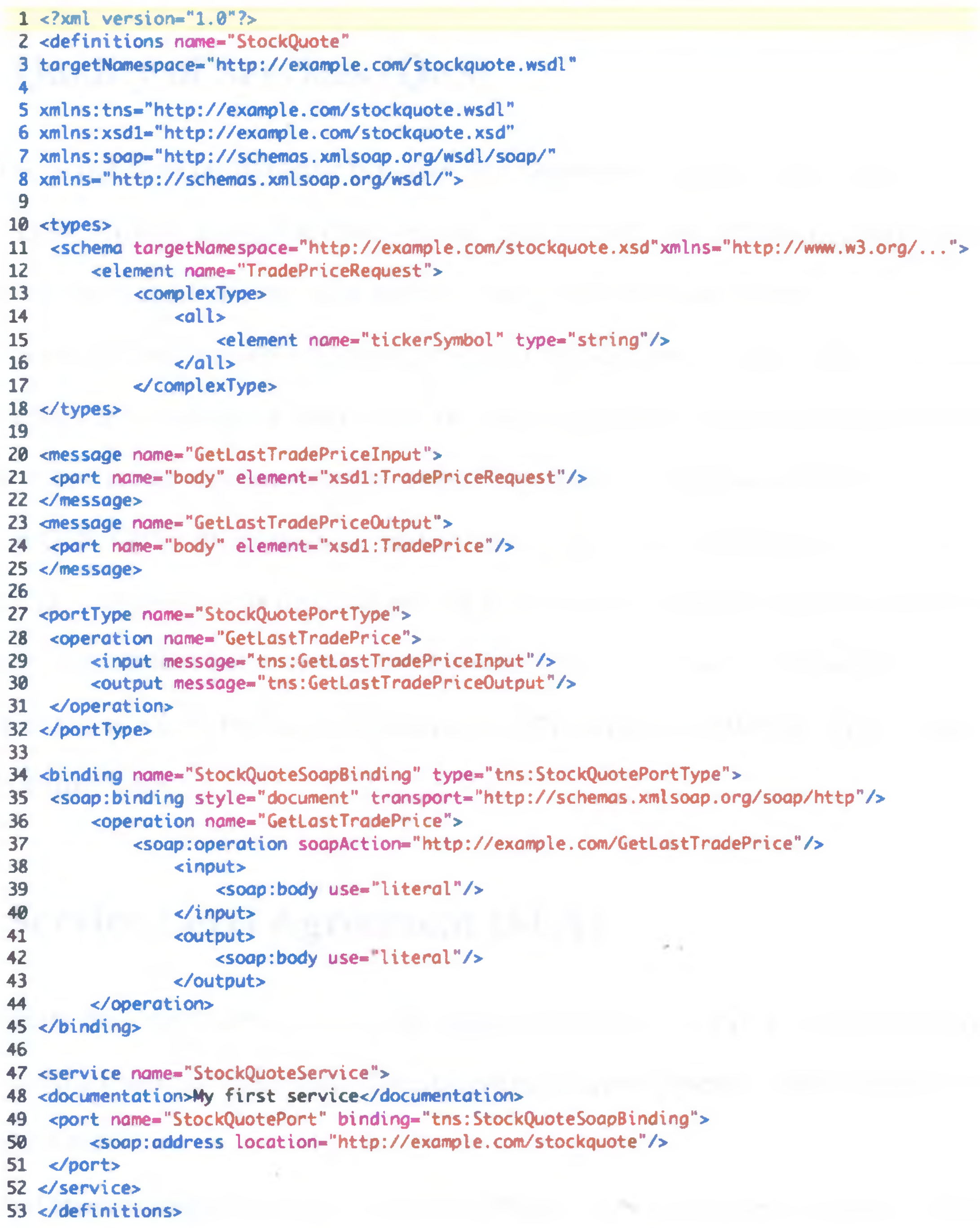
Figure 2.2: SOAP response message example [3]

2.2.3 WSDL

Web Services Description Language (WSDL) is an XML-based shared schema that describes interfaces and interactions within services [9]. Any WSDL document has seven attributes: *Types*, *Message*, *portType*, *Operation*, *Binding*, *Port*, and *Service*. To illustrate WSDL usage, the WSDL of StockQuote service is presented in Figure 2.3 [12] . The Types attribute is a container for data type definitions using some type system (line 10 - 18). The Message attribute is a definition of the sent data (line 20 - 25). The portType attribute is set of operations supported by services' port (line 27 - 32). The Operation attribute refers to input messages and output messages. Four basic operations can be defined in any WSDL document [2]. The request-response operation is used in the given example (line 28 - 31). The Binding attribute is a definition of service implementation for a particular portType (line 34 - 45). The Service attribute is a combination of related ports (line 47 - 52). The Port attribute is a single communication endpoint defined as a combination of a binding and a network address (line 49 - 51) [4].

2.2.4 UDDI

UDDI (Universal Description, Discovery, and Integration) is a set of service registries that are used as brokers between users and service providers [9]. UDDI is similar to yellowpage books that direct people to needed services' information, such as, local hospital phone. UDDI provide users with necessary information about deployed services in an intended composition. UDDI



```
<?xml version="1.0"?>
<definitions name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"

xmlns:tns="http://example.com/stockquote.wsdl"
xmlns:xsd1="http://example.com/stockquote.xsd"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
 <schema targetNamespace="http://example.com/stockquote.xsd"xmlns="http://www.w3.org/...">
     <element name="TradePriceRequest">
         <complexType>
             <all>
                 <element name="tickerSymbol" type="string"/>
             </all>
         </complexType>
</types>

<message name="GetLastTradePriceInput">
 <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
 <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
 <operation name="GetLastTradePrice">
     <input message="tns:GetLastTradePriceInput"/>
     <output message="tns:GetLastTradePriceOutput"/>
 </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
 <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="GetLastTradePrice">
         <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
             <input>
                 <soap:body use="literal"/>
             </input>
             <output>
                 <soap:body use="literal"/>
             </output>
     </operation>
</binding>

<service name="StockQuoteService">
<documentation>My first service</documentation>
 <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
     <soap:address location="http://example.com/stockquote"/>
 </port>
</service>
</definitions>
```

Figure 2.3: WSDL document example [4]

locates services and provides access to their WSDL documents. This information is important to users because UDDI allows users to interact with their suitable services.

2.3 Quality of Services (QoS)

In order to understand the dynamic behavior of compositions, the run-time behavior of the services in compositions should be observed and understood. The following Section presents a brief introduction about the run-time behavior properties of composition.

It is possible that for a service that there are several service instances that can be used. Service instances may distinguish themselves by making promises about run-time behavior. Run-time behavior may be characterized by a set of attributes (or metrics) referred to as *Quality of Service* or QoS. QoS metrics include: performance, reliability, scalability, capacity, robustness and so on [13]. A promise is made about that the behavior, as characterized by a QoS metric, will satisfy a condition. For example, *service time*, which is a time needed to process a request from a service, is a QoS performance metric, and the promise is that the service time will be less than x time units.

2.4 Service Level Agreement (SLA)

To determine the responsibilities and expectations of the services in a composition an agreement or contract should be initiated. The following Section presents several aspects related to these agreements.

A set of promises agreed upon between a client and service provider is referred to as *Service Level Agreement* in order to execute a business process, which is an action taken in the course of conducting business. Each promise is referred as *service level objectives*(SLO). Any SLO has a functional part and a guarantee part. The functional part refers to a system, endpoint, or a process. The guarantee part involves a particular instance of the agreement that will be applied on the functional part of the SLO. For example, manufacturing PCs company (PCMaker.com) and a company buying PCs company (PCBuyer.com) initiated an SLA for a period of x time with several SLOs. One SLO of this contract would be "PCMakers e-procurement system will be available to PCBuyer1, Monday to Friday from 9AM-5PM, 99.9% of the time". SLOs play a crucial role in SLA life cycle [14]. If the condition specified in the SLO is not satisfied at run-time then SLO is considered to be violated.

Several standards have been developed in recent years for negotiating and representing formal SLAs. WSLA [9, 15], WS-Agreement [16], WSOL [9], and SLAng [17] are few examples to count. There is not yet a SLA specification language that is considered as an official modeling language because most languages are designed and used to fit certain requirements [15].

2.5 Management

In order to offer a clear view of how services of compositions perform, management that observes and reacts to faulty actions within compositions should be deployed. The following Section introduces several aspects related to management.

Management entails the operation, administration and maintenance of a computing system so that the system behaves as expected with respect to availability, performance and security. The management of a service composition spans a range of activities that includes monitoring of the run-time behavior of a service, analysis of monitored data, and determining recovery actions to modify the run-time behavior [18]. The monitoring activity typically consists of periodically monitoring the on-line status of services. Monitoring is an essential part of management. The management analysis is concerned with the causes of why services do not satisfy the expectation specified in a SLA. Decision making components are leveraged to perform *recovery actions*. Recovery actions should benefit the affected party by informing with the source of the violation or suggesting with alternative providers or services if applicable [19].

2.6 Monitoring

To improve the performance of a composition, monitoring of QoS measurements of intended services either that show normal or abnormal behaviours is a necessity. The following Section introduces several aspects related to such monitoring.

Two typical monitoring mechanisms are mentioned in the literature - *message interception* [8, 19] and *code level instrumentation* [8]. Message interception mechanism intercepts exchanged requests and responses messages between services' compositions and the clients. Message interception is used as two styles. The first style is that standalone internal agents are embedded within messaging framework at host environments. This style requires installed agents for each individual web service in a composition and gives management capabilities to the host of these services [20]. The second style is that a decoupled external intermediary or third party component is located between clients and services' composition. The third party has its sensors that monitor run-time behaviors of services and record QoS metrics of a managed service. By this style, the third party has better visibility and control over each of the services [21]. Although message interception is a common mechanism and offers easy maintenance, it suffers from management complexity, possible bottlenecks, and points of failure [8]. Code Level Instrumentation refers to code that is place in the code of the service that provides various monitoring and reporting functions about these services. Code Level Instrumentation can provide extensive and accurate monitoring data but is costly to build. Further discussion about code level instrumentation mechanism is provided in [8].

2.7 Fault Diagnosis

A *fault* (known as a problem(P)) is a defect in a software or software component [22]. A system is said to have a *failure* if the service it delivers to the user deviates from compliance with the system specification for a specified period of time [23]. A problem causes a failure. To illustrate these concepts consider the following examples: (1) A hardware power loss causes a web service to become unavailable (i.e, physical problem leads to failure); (2) An unexpected load causes a web service to violate its SLA (i.e, operational problem leads to failure). Failures are often referred to as *symptoms* of a problem. The term symptom (S) is often used interchangeably with failure and an *event* is defined as a notification of a failure. A problem can occur on one component but a failure is detected on another component. The two examples presented earlier show this. The hardware power problem occurs at the server side but the failure that is the result of this fault is detected by a process on a different machine. This suggests a need to be able to determine a problem based on failures. This is referred to as fault diagnosis.

Because problems are unavoidable and may pose critical impacts on compositions (i.e, problems may delay system functionalities or may terminate processes) [24], their quick diagnosis is essential to maintain the robustness, reliability, and accessibility of a system [5].

The process of fault diagnosis usually involves three steps: *fault detection*, *fault localization*, and *testing* [5]. Fault detection is a process of capturing symptoms arising from the affected system [24]. Detection techniques can be based on active schemes (e.g, polling) or symptom-based schemes, where a system component indicates that it has detected a failure. Several fault detection techniques are proposed, e.g, Angeli et al [25], Hwang et al [26]. Fault localization requires an analysis of a set of observed symptoms. The goal of fault localization is to find an explanation of the symptoms' occurrence. The explanations are delivered in the form of *hypotheses*. Hypotheses are statements which explain that each observed symptom is caused by one or more designated problems. The validity of a hypothesis is evidenced by the efficiency of its fault diagnosis. Based on these hypotheses, a testing step is performed in order to determine the actual problems through the application of a suitable testing mechanism [5].

2.8 Related Work

The following Sections present common event correlations and some previous work on fault diagnosis.

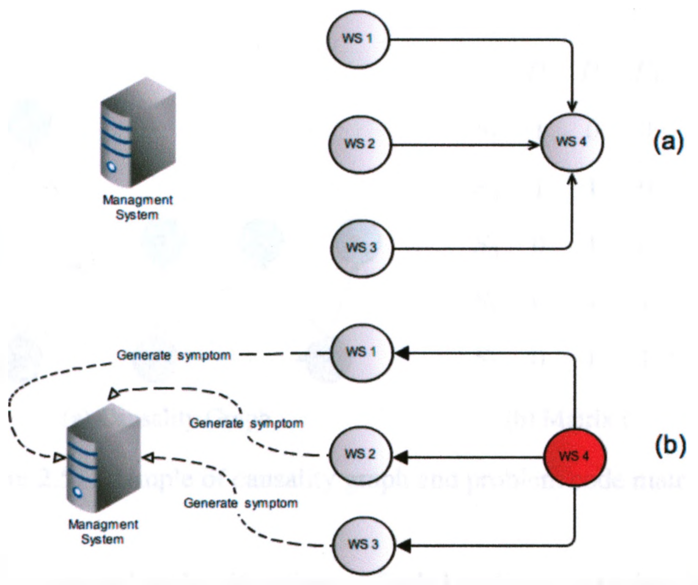
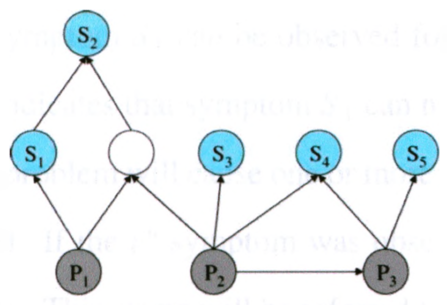


Figure 2.4: Compression correlation applied in a system of four web services

2.8.1 Event Correlation

Event correlation is a technique that can be used for the fault localization process. Event correlation attempts to associate one symptom with another symptom in order to infer the relationship between their occurrences [27]. Through an examination of these associations, a number of possible hypotheses are generated that reflect the symptoms' occurrence.

There are several different types of correlations, which are useful for diagnosing problems in a network. Compression correlation, for example, reduces multiple occurrences of the same symptom into a single symptom [27]. Compression correlation can be used for an application that consists of a set of services. For example, an application may consist of four interacting web services WS_1, WS_2, WS_3, and WS_4. WS_1, WS_2, and WS_3 make requests of WS_4. This is presented in Figure 2.4.a. If the machines that hosts WS_4 has a power loss problem, WS_1, WS_2, and WS_3 would generate symptoms indicating that WS_4 is being slow or unavailable. This is depicted in Figure 2.4b. Since a management system has knowledge about the workflow of these services, the management system could apply compression correlation in order to reduce the multiple occurrences of the redundant symptoms from WS_1, WS_2, and WS_3.



(a) Causality Graph

$$\begin{array}{c|ccc} & P_1 & P_2 & P_3 \\ S_1 & 1 & 0 & 0 \\ S_2 & 1 & 1 & 0 \\ S_3 & 0 & 1 & 0 \\ S_4 & 0 & 1 & 1 \\ S_5 & 0 & 1 & 1 \end{array}$$

(b) Matrix C

Figure 2.5: Example of causality graph and problem code matrix [5]

Steinder et al [5] proposed a classification of fault localization techniques which is derived from Graph-theoretic techniques, Artificial Intelligence, and Model-traversing techniques. Graph-theoretic techniques rely on a graphical model. This type of graphical model is referred to as a *fault propagation model* (FPM). FPM is a graph that include symptoms, problems, and the relationship between them. This type of graph is an example of a causality graph. In such graphs, endpoints may be marked as being solely problems or being solely symptoms, while others may be marked as problems and symptoms at the same time. Edges describe cause-effect relationships between problems and symptoms or symptoms and other symptoms. An example is seen in Figure 2.5a. To create such a model, an accurate knowledge of current dependencies among the system components is required. Codebook, context-free grammar, and bipartite causality approaches are a few examples of Graph-theoretic techniques [5].

The codebook technique [5] uses a matrix representation of causality graph[1] in order to infer the causes of observed symptoms. A causality graph is a bipartite graph whose vertices can be partitioned into two disjoint subsets V and W such that each edge connects a vertex from V to one from W [28]. The matrix is referred to as *problem codes matrix* (PCM)[2] and is built based on the causality graph. An example of the causality graph and the problem codes matrix (say matrix C) are illustrated in Figure 2.5a and 2.5b, respectively. The matrix

[1]For system analysis purposes, a causality graph is considered as an efficient knowledge base

[2]Problem codes matrix term equals to the codebook term

consists of a column that represents symptoms that problems cause. A matrix entry either has the value of zero or one. For example, the value of one assigned at $C[1,1]$ position in matrix C indicates that symptom S_1 can be observed for problem P_1. The value of zero assigned at $C[1,3]$ position indicates that symptom S_1 can not be observed for problem P_3.

At run-time a problem will cause one or more symptoms to be generated. From this a string can be formulated. If the i^{th} symptom was observed then the i^{th} position in the string is one otherwise it is zero. This string will be referred to as a *current symptoms vector* (CSV).

The diagnosis process uses the *Hamming distance*. The Hamming distance is the minimum *number of substitutions that transforms one string into the another.* For example, the Hamming distance between two words "toned" and "roses" is three letters and the Hamming distance between the two strings 1011101 and 1001001 is two bits [29]. Each value in a column in the matrix is compared with its corresponding code in a given CSV. If both values are identical (i.e, the value in the column in the matrix and its corresponding code in the given CSV are the same), the Hamming distance value is denoted as zero. Otherwise, the Hamming distance is denoted as one. The values are then summed to determine the Hamming distance of the two words. The minimum of the Hamming distance values is an indicator of the corresponding problems as the causative problems. For the matrix C, if the given CSV is 11000, the Hamming distance is (0,4,4). Thus, the causative problem was P_1. If the given CSV is 11101, the Hamming distance is (2,2,4). Thus, the causative problems are limited to P_1 and P_2.

Since the coding phase is performed only once, the codebook approach is very fast, robust, and efficient. On the other hand, the accuracy of the codebook technique is hard to predict when more than one problem occurs with overlapping sets of symptoms. In addition, since each change of system configurations requires regenerating the codebook, the technique is not suitable for environments with dynamically changing dependencies [5].

2.8.2 Diagnosis in Policy-based Management System

Tighe [30] implemented a distributed fault diagnosis algorithm, proposed by Peng and Reggia and is referred to as *Parsimonious Covering theory* [6, 31], in a policy-based management tool called BEAT (Best Effort Autonomic Tool) [32, 33]. In this context, Tighe used the terms disorder and manifestation instead of problem and symptom, respectively. The algorithm is concerned with the generation of plausible hypothesises, based on given information that comes from graph-theoretic models, prior to diagnosis. Hypotheses are delivered and grouped in order to generate disorder-and-manifestation statements that are forwarded to a decision making system for recovery actions.

Tighe used the algorithm to generate hypothesises based on a simple bipartite (disorder-and-manifestation) graph. The disorder-and-manifestation graph, which is referred to as a causal network, consists of a set of vertices representing underlying disorders and manifestations and a set of arcs or edges representing the causal relationship between the two. All disorder vertices are directed to manifestation vertices. Therefore, a disorder vertex causes or covers a set of manifestation vertices and the manifestation vertices are caused or covered by the disorder vertex. The presence of a manifestation vertex can evoke or suggest that all of its linked disorders vertices possibly are causatives [6]. An example of the causal network is depicted in Figure 2.6.

In order to understand the Peng and Reggia algorithm, a few notations should be declared in the first place. Peng and Reggia [31, 6] used $\langle D, M, C, M^{+} \rangle$ notation to represent their algorithm. $D = \{d_1, \ldots, d_n\}$ is a finite non-empty set of disorders and n is the number of disorders. $M = \{m_1, \ldots, m_n\}$ is a finite non-empty set of manifestations where n is the number of manifestations. $C \subseteq D \times M$ is a relation with $domain(C) = D$ and $range(C) = M$. The causal relationship between sets D and M is contained in C with $\langle d_i, m_i \rangle \in C$ iff d_i causes m_i. M^{+} is a subset of M that represents a set of present manifestations. In general, the algorithm is deployed in the form of *abductive reasoning*. Abductive reasoning most closely resembles how a human being diagnoses problems. A problem consists of a set of rules, a specific case, and a

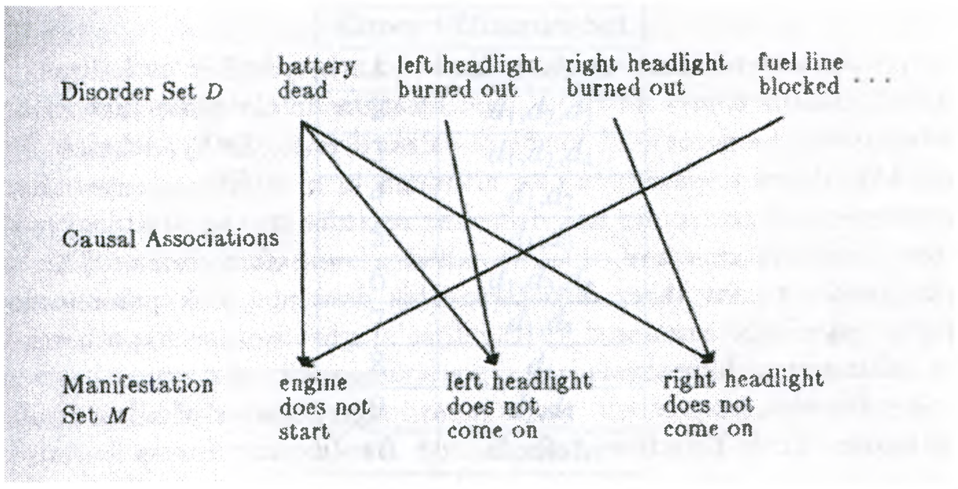

Disorder Set D
battery dead
left headlight burned out
right headlight burned out
fuel line blocked ...
Causal Associations
Manifestation Set M
engine does not start
left headlight does not come on
right headlight does not come on

Figure 2.6: Causal network for automotive problems [6]

result that occurs given the relationship between the two. C represents the set of rules. D represents all possible specific cases or disorders. M represents all possible result or manifestations. M^+ represents a designated set of manifestations of the current problem. The set of rules and the result are already known. What is needed is to hypothesize about the specific case that is causing the result. To illustrate the concept of abductive reasoning, when a doctor diagnoses a patient, the set of manifestations experienced by the patient would be analogous to the result and the doctor's medical knowledge would be the set of rules. Thus, the doctor's diagnosis would indicate to the set of specific cases [30].

As mentioned earlier, the algorithm [6] concerns about generating a set of hypothesises or *covers*. A *cover* of M^+ is a set disorders $D_I \subseteq D$ such that each manifestation in M^+ can be caused by at least one disorder in D_I. The process of *covers* generation is about encapsulating M^+ with possibly causative disorders in D_I. The source of such causality relationship is based on the graphical model of the causal network. Each *cover* represents a single hypothesis that gives potential explanations for each manifestation in M^+. It is logically that if present manifestations M^+ equals M, the generated *covers* should cover all manifestations in M. This fact makes the proposed algorithm very effective, for single disorder and manifestation, or might not be effective, for multiple disorders and manifestations. For that reason, Peng and Reggia [6, 31] suggested that simple or parsimonious *covers*, which have fewer disorders, are more

Cover	Disorder Set
1	d_1,d_2,d_3,d_4
2	d_1,d_2,d_3
3	d_1,d_2,d_4
4	d_1,d_2
5	d_1,d_3
6	d_1,d_3,d_4
7	d_1,d_4
8	d_1
9	d_2,d_3
10	d_2,d_3,d_4

Table 2.1: All *covers* set disorders D_I of set manifestation M

likely to be true. They also suggested different criteria for judging the simplicity. Inferring more simple *covers* will help to narrow down the broad spectrum of disorders to a satisfied limit. The simplicity criteria includes: *Minimal*, *Irredundant* and *Relevant covers*, and etc. A *Minimal cover* is a *cover* D_I of M_J, since $M_J \subseteq M$, that the *cover* contains the minimal number of disorders required to cover M_J. An *Irredundant cover* is a *cover* D_I of M_J where each disorder causes at least one manifestation that no other disorders in the same *cover* causes. A *Relevant cover* is a *cover* D_I of M_J where each disorder causes at least one manifestation with considering that two or more disorders may cause the same manifestation in the same *cover*. The set of *Minimal covers* for a set of manifestations is a subset of *Irredundant covers*, which is a subset of the set of *Relevant covers*. Because the criteria create increasingly broad sets of covers as moving from *Minimal* to *Relevant covers*, a sequence of *covers* filtering should be applied [30].

Figure 2.6 depicts an example given by Peng and Reggia [6, 31] that describes the diagnosis of automotive problems. The disorders include *battery dead* (d_1), *left headlight burned out* (d_2), *right headlight burned out* (d_3), and *fuel line blocked* (d_4), which are the upper-side vertices. The manifestations are *engine does not start* (m_1), *left headlight does not come on* (m_2), and *right headlight does not come on* (m_3), which are lower-side vertices. All covers D_I of M^+ are generated in table 2.1 considering that $M^+ = \{m_1, m_2, m_3\} = M$. It is obvious that all current *covers* do not indicate the causative disorders and will not help to generate a plausible hypothesis. The goal is to reduce the number covers in order to gain simple covers. For example, say that $M^+ = \{m_2, m_3\}$. In order to get simple *covers*, any disorder must cause at least one manifestation in M^+. From table 2.1, it is obvious that there are five *covers*, that is, *Relevant* and cover M^+. These *covers* are 2,4,5,8, and 9. Others *covers* are non-useful because they entailed d_4 which does not cause any manifestation in M^+. It is obvious that such *covers* are not simple enough to generate a plausible hypothesis. In order to gain more simple *covers*, any disorder must cause at least one manifestation in M^+ that is not caused by any other disorder in the same *cover*. From table 2.1, it is obvious that there are two *covers*, that is, *Irredundant* and still cover M^+. These *covers* are 8 and 9. *Cover* 2,4,5 are not needed because they entailed redundant disorders. *Cover* 8 and 9 may considered as a plausible hypothesis, but it is better to have very simple *covers*. In order to gain more simple *covers*, at least one manifestation in M^+ must be caused by the possible fewest number of disorders. From table 2.1, it is obvious that there is one *cover*, that is, *Minimal* and still covers M^+. This *cover* is 8. *Cover* 9 is excluded because the disorders of M^+ could be covered by *cover* 8 alone. Therefore, *cover* 8 or both 8,9 covers seem to be reasonable hypotheses.

Although producing a hypothesis that explains the occurrence of manifestations would help in fault diagnosis, it is impossible to guarantee that a definitive diagnosis can be obtained because determining which hypothesis is correct or more likely to be correct is a complex task.

2.8.3 Diagnosis by Fault Taxonomies

Other work on fault diagnosis [34, 35] state that only knowing what faults to look for is a convenient approach to suggesting a suitable recovery mechanism more quickly by building fault taxonomies that explicitly indicate symptoms and problems. Therefore, system administrators are educated on how to react or treat present problems and might be capable of handling future problems as soon as they are faced with the listed symptoms in the fault taxonomies. However, some problems are unpredictable and might spread to a new composition. Knowing the taxonomies alone will not identify the problems in the future compositions. Therefore, to some degree, fault taxonomies are not the optimal approach to fault diagnosis.

2.9 Summary

This chapter covered basic information necessary to build a context for the following chapters and to present related research on fault diagnosis.

Chapter 3

Proposed Approach

This Chapter is concerned with mapping of a high-level specification of a web services composition to a diagnostic model. This mapping can be done by a third party service that is not part of the application resulting from the composition of the web services. This process should be automated in order to reduce errors.

Fault localization is a process of deducing the source of a failure from a set of observed symptoms. In the previous Chapter we presented approaches based on graph-theoretic techniques. These techniques require a priori specification of a failure condition in component is related to failure conditions in other components. In this Chapter we show how a specification of a composition of web services can be used as the a priori specification.

This Chapter is organized as follows: Section 3.1 describes the high-level specification of a web services and some of its properties in more details. Section 3.2 shows two business processes examples that are used to illustrate the proposed approach. Section 3.3 shows the mechanism used for the high-level specification of a web services composition mapping. Section 3.4 shows a diagnostic model used to represent the diagnosis method used in our research.

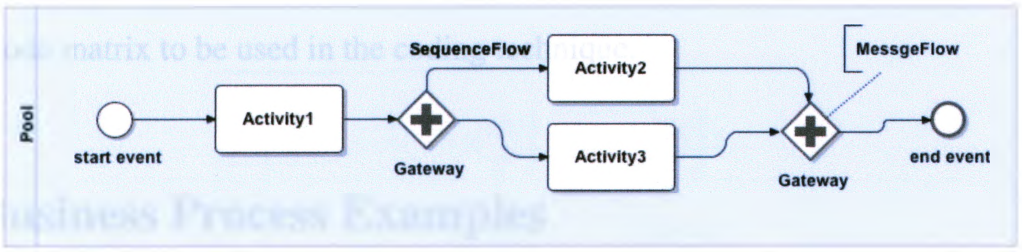


Figure 3.1: Simple BPMN example

3.1 BPEL

This Section focuses on the high-level specification of a web services composition. BPEL is a standard for an XML-based language for describing the interaction between the participants in a process, its operational logic and execution flow. BPEL specifications can be quite complex and hence there are a number of tools that allow users to conceptualize business processes as directed graphs. One example is Business Process Modelling Notation (BPMN). Notational elements in BPMN include FlowObjects which are contained in pools. One type of FlowObject represents an activity [1] [36]. An activity can either be atomic or compound where a compound activity is structured from other activities. An activity FlowObject is a node that may have multiple outgoing links representing different possible flows. The outgoing link chosen depends on the result of the evaluation of a condition. Decision points or Gateways represent these conditions. The link from a Gateway node to an activity node is referred to as a SequenceFlow. MessageFlows describe the exchange of messages between pools [36]. Another type of FlowObject denotes Events the start or end of a flow. A pool consists of a composition of FlowObjects, Gateways, and SequenceFlows and MessageFlows. A pool may have an activity FlowObject that can be represented by another pool. Each pool represents a workflow and a business process is associated with a set of pools. An example of a business processes workflow modelled as a BPMN specification is presented in Figure 3.1.

We can use the specification of a business process using BPMN to generate a composition

[1] An activity represents a web service

dependency graph. This will be used to generate a causality graph which is the basis for the problem code matrix to be used in the coding technique.

3.2 Business Process Examples

This Section shows two business processes examples: loan business process, and office business process.

3.2.1 Loan Business Process

A BPMN model of the loan business process composition is depicted in Figure 3.2 [7]. This business process has three players: client, bank composition, and credit company composition. Three pools are presented in Figure 3.2, Figure 3.3, Figure 3.4: *LoanProcess*, *CheckLoanApplicationInformation*, and *MakeDisbursement*, respectively. The first and the third pools represent the bank composition and the second pool represents the credit composition. The LoanProcess pool is the main pool in this business process.

In the LoanProcess pool, the workflow of the loan process is triggered as soon as loan application forms are received from clients after the forms have been filled. All loan application forms from other branches are gathered and submitted to establish the loans requests. The loan requests will be sent to the second pool, CheckLoanApplicationInformation pool, for verification purposes. The verification step indicates if a client has a bad credit or has a good credit. The results of the verification are returned to the LoanProcess pool in order to make disbursement decisions or rejection decisions with the justification for rejection. If the results of the verification were negative, the decision is made to reject the loan requests. Before the business process workflow is terminated, the clients are informed about the rejection. If the results of the verification were positive, the LoanProcess pool notifies the client that their loan request approved and forwards the loan request to the MakeDisbursement pool in order to finalize how the loans are disbursed. After the disbursement step, the workflow is completed.

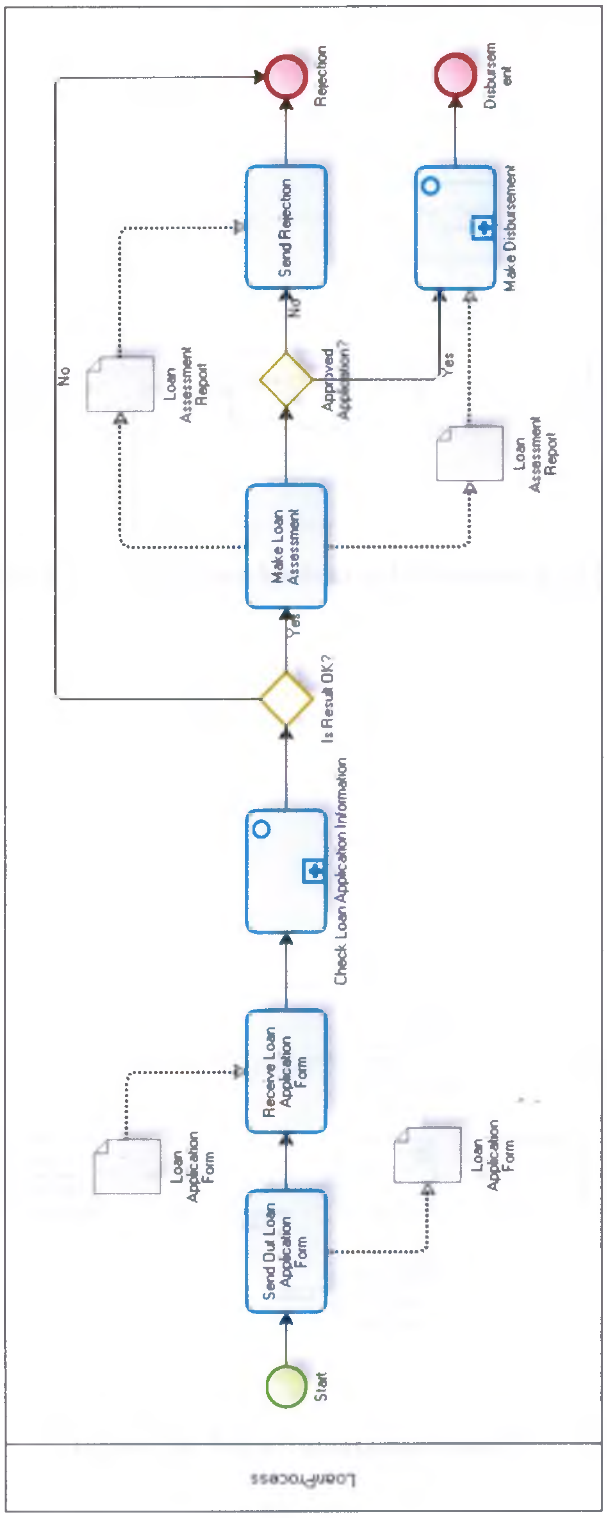


Figure 3.2: LoanProcess pool [7]

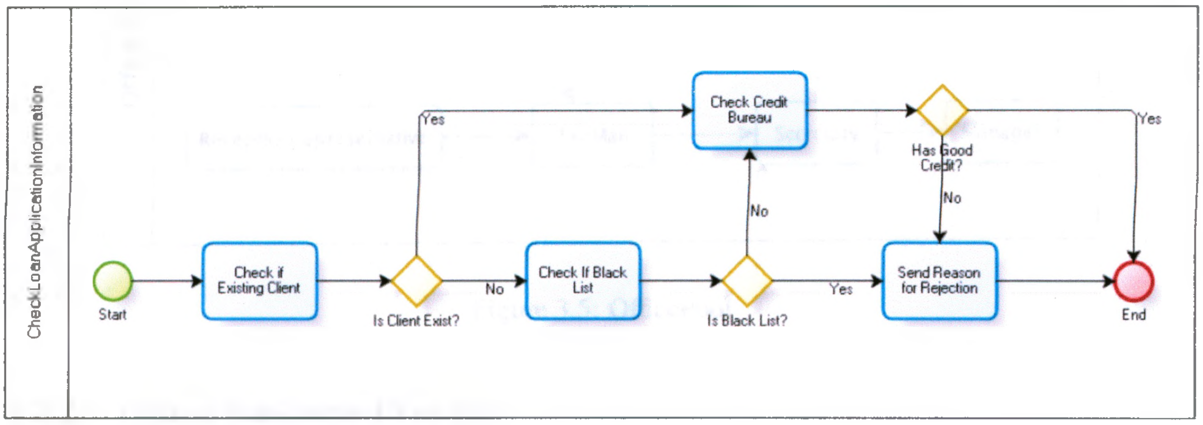


Figure 3.3: CheckLoanApplicationInformation pool [7]

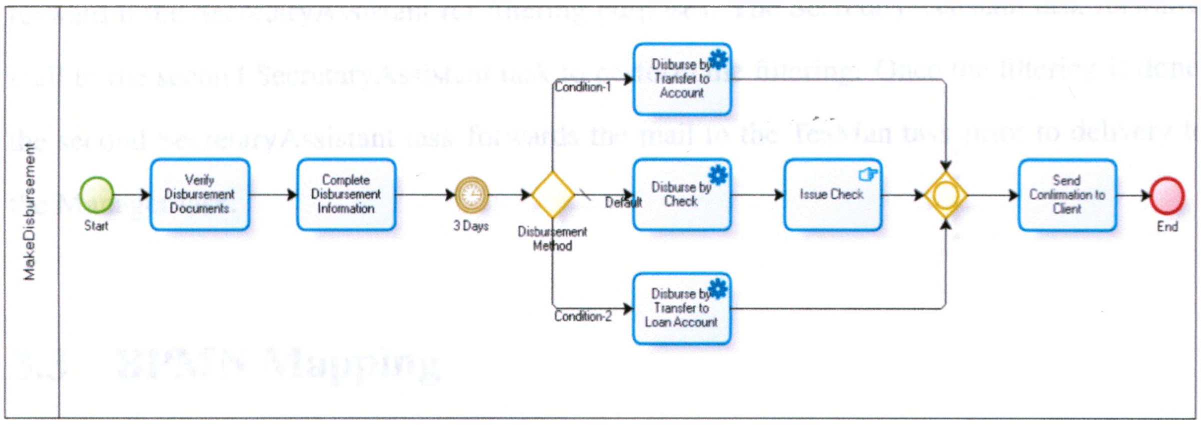

MakeDisbursement
Start
Verify Disbursement Documents
Complete Disbursement Information
3 Days
Disbursement Method
Condition-1
Default
Condition-2
Disburse by Transfer to Account
Disburse by Check
Issue Check
Disburse by Transfer to Loan Account
Send Confirmation to Client
End

Figure 3.4: MakeDisbursement pool [7]

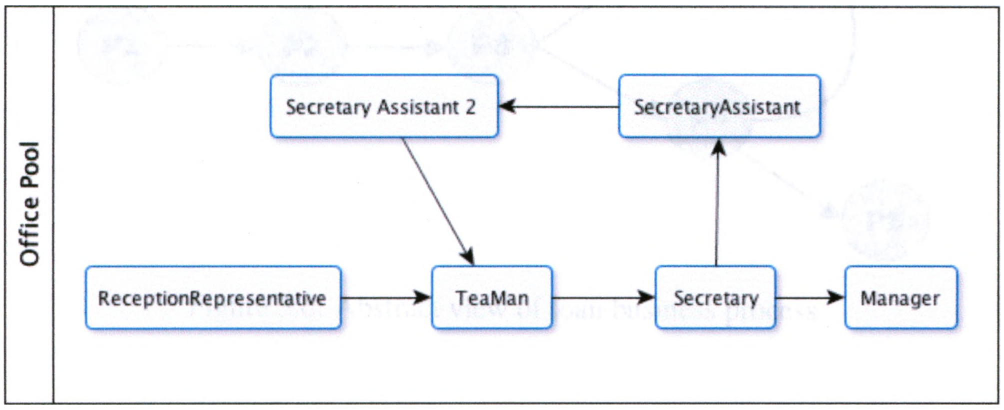


Figure 3.5: OfficePool

3.2.2 Office Business Process

Another business process example is an office process, which is depicted in Figure 3.5. In the OfficePool, the workflow of the office business process is triggered as soon as office mails are received by the ReceptionRepresentative task. The TeaMan task takes the mails from ReceptionRepresentative task and passes it to the Secretary task in order to be filtered and passed to the Manager task. If there are urgent mails, the Secretary task will forward it to the Manager task directly. If there is non-urgent mail, commercial, or spam mails, the Secretary task will forward it the SecretaryAssistant for filtering purposes. The SecretaryAssistant task forwards mail to the second SecretaryAssistant task to perform the filtering. Once the filtering is done, the second SecretaryAssistant task forwards the mail to the TeaMan task prior to delivery to the Manager task.

3.3 BPMN Mapping

This Section describes how a BPMN mapping of a web services composition is performed. The BPMN mapping is done through the transformation from BPMN graphs to a composition dependency graph (CD) which is done prior to determining the causality graph.

The transformation from BPMN to CD is performed as follows: assume that CD is repre-

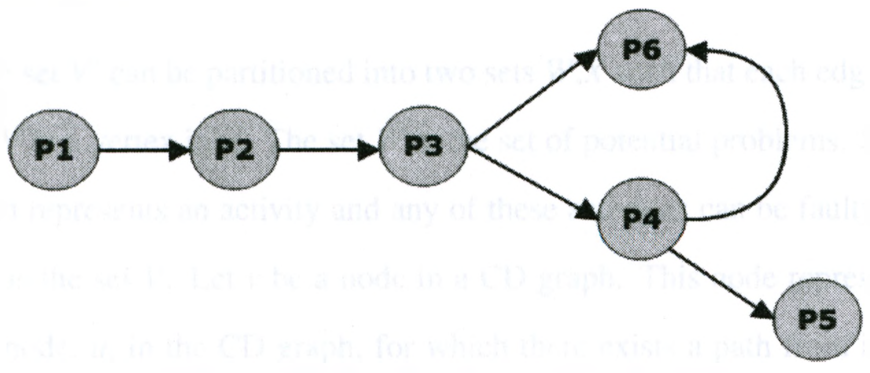


Figure 3.6: Abstract view of loan business process

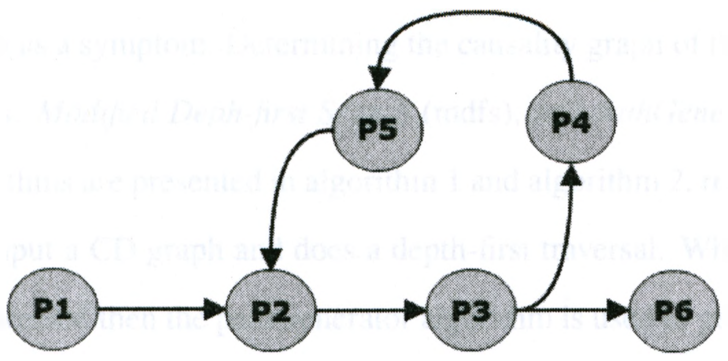


Figure 3.7: Abstract view of office business process

sented as (V,E). Each BPMN atomic activity node is a node in V. If a decision point follows an activity then the node in V representing the activity will have two outgoing edges. Edges represent different possible flows. The CD graph for the LoanProccess pool process is depicted in Figure 3.6, where P1 represents SendOutLoanApplicationForm task, P2 represents Received-LoanApplicationForm task, P3 represents CheckLoanApplicationInformation subprocess, P4 represents the MakeLoanAssessment task, P5 represents the MakeDisbursement subprocess and P6 represents the SendReject task. P3 and P5 represented subprocesses each with its own set of activities. The CD graph for the office business process is depicted in Figure 3.7, where P1 represents ReceptionRepresentative task, P2 represents TeaMan task, P3 represents Secretary task, P4 represents the SecretaryAssistant task, P5 represents the SecretaryAssistant2 task and P6 represents the Manager task. As can be seen the CD graph is an abstract view of a BPMN process.

Assume that the CD graph is represented as (V,E) while the causality graph is represented as (V',E')[2]. The set V' can be partitioned into two sets W,X such that each edge in E' connects a vertex from W to a vertex in X. The set W is the set of potential problems. Since each node in the CD graph represents an activity and any of these activities can be faulty then the set of W is the same as the set V. Let v be a node in a CD graph. This node represents a potential problem. Any node, u, in the CD graph, for which there exists a path from it to the node v, potentially could exhibit a failure condition if v becomes faulty. Any node that could exhibit a failure condition is in set X. For a node u we use the notation P_u to represent u as a problem and S_u to represent u as a symptom. Determining the causality graph of the CD graph requires these two algorithms: *Modified Deph-first Search* (mdfs), and *pathGenerator*. The mdfs and pathGenerator algorithms are presented in algorithm 1 and algorithm 2, respectively. The mdfs algorithm takes as input a CD graph and does a depth-first traversal. When all child nodes of node v have been traversed then the pathGenerator algorithm is used to generate all paths from node v to each leaf node. These paths are used to produce the causality graph. The causality graph of the loan business process is depicted in Figure 3.8. The causality graph of the office business process is depicted in Figure 3.9.

The mdfs algorithm uses two variables: *VerticesList*, and *BackTrackEdgesList*. VerticesList is a list that keeps track of each node's label. The BackTrackEdgesList maintains a list of backtrack edges. A backtrack edge (v,w) indicates that the mdfs algorithm is revisiting node w and that not all of node w's children had yet been visited. *White* is a label that indicates an unvisited node, which is the initial state for all nodes. *Gray* is a label that indicates a node has been visited but not all of its children have been traversed. *Black* is a label that indicates a node has been visited and all of its children have been processed. When the input CD graph is received, mdfs is triggered (line 1). If the current node being visited is White, mdfs will assign the Gray label (line 3). The mdfs algorithm examines each outgoing edge (lines 4-5). If the node on the other end of the edge is labelled White then this means that the node has not been visited and thus no paths have been generated (lines 6-7). If the node on the other end of the

[2]The causality graph vertices are known in advance based on the given information from a client about fault and symptom quantities

Algorithm 1: Modified depth-first search(mdfs)

```
Procedure: mdfs {executed on receipt Graph G with root node v}
Input     : $G = (V, E)$ where $E = \{(v, w) | v, w \in V\}$ and node $v$ is a zero indegree edge
            and all nodes $v$ are initially unvisited.
Variables : VerticesList carrys on all nodes, White is label for unvisited node state,
            Gray is label for the visited but not finished node state. Black is label for the
            finished node state. BackTrackEdgesList carrys on edges resulted from
            visiting Gray nodes.
1  mdfs(G,v)
2  if VerticesList [$v$] = $White$ then
3      VerticesList [$v$] = $Gray$
4      forall the $e \in G.incidentEdges(v)$ do
5          w = $G.incidentEdges(v, e)$
6          if $VerticesList[w]$ = $White$ then
7              mdfs($G, w$)
8          else if $VerticesList[v]$ =$Gray$ then
9              putEdge(v,w,BackTrackEdgesList)
10     VerticesList [$v$] = $Black$
11     // when there are zero unvisited nodes, backtrack
12     pathGenerator(v)
```

edge is labelled Gray then the edge is put in the BackTrackEdgesList (lines 8-9). If there is no unvisited neighbour node for the current node, mdfs executes the pathGenerator algorithm in order to generate paths (line 12).

The pathGenerator algorithm is executed when all nodes on the other end of the outgoing edges of node v have been visited. The pathGenerator uses three variables: `newPath`, `pathsW`, and `Paths`. The `newPath` variable is used to represent a sequence of nodes, and `pathsW` represents a set that contains all the paths from w to all leaf nodes. `Paths` is a container for all possible paths. pathGenerator algorithm is executed when a current node v is received from mdfs. The pathGenerator looks for outgoing edges of node v. If there are no outgoing edges (line 2), the pathGenerator algorithm creates a new path, appends v node in this path, and adds the path to `Paths` (lines 5-7). If there are one or many outgoing edges (line 8), the pathGenerator algorithm retrieves each path associated with w and creates a new path by putting together v and the path associated with w (lines 10-19).

Algorithm 2: pathGenerator

```
Procedure: pathGenerator {executed on receipt a graph G and node v}
Input     : Graph $G$ and node $v$ from mdfs
Variables : newPath, pathsW, and Paths
Output    : Possible set of paths
begin
    if $G.incidentEdges(v) == null$ then
        // Create a new path, add $v$ node in this path,
        // and add the path to Paths.
        newPath = null
        newPath.append($v$)
        Paths = Paths ∪ newPath
    else
        forall the $e \in G.incidentEdges(v)$ do
            w = $G.incidentEdges(v, e)$
            pathsW = emptySet
            // Retrieve all previously generated paths from
            // $w$ to each leaf node reachable from $w$
            forall the $p \in$ Paths.$get(w)$ do
                newPath = null
                newPath.append($v$)
                newPath.append(p)
                pathsW.add(p)
            Paths = Paths ∪ pathsW
```

The execution of the algorithms does not always provide all paths. This happens where there is a cycle. The existence of backtrack edges indicate a cycle. Assume a backtrack edge: (*v*,*w*). The mdfs algorithm will generate all paths from node *w* to leaf nodes but the paths generated for node *v* will not include those paths that start at *w*. For example, if the edge (*P5*,*P2*) is a backtrack edge in the office business process (see the Figure 3.9). The paths from the root node ($P1$) to all nodes in the office CD graph are: (($P1$), ($P1,P2$), ($P1,P2,P3$), ($P1,P2,P3,P6$), ($P1,P2,P3,P4$), ($P1,P2,P3,P4,P5$)). After considering the backtrack edge (*P5*,*P2*), the paths will be: (($P1$), ($P1,P2$), ($P1,P2,P3$), ($P1,P2,P3,P6$), ($P1,P2,P3,P4$), ($P1,P2,P3,P4,P5$), ($P1,P2,P3,P4,P5,P2$)). Paths generated considering backtrack edges are done after mdfs terminates. Let (*v*,*w*) be a backtrack node. Let *P* be the set of paths. For

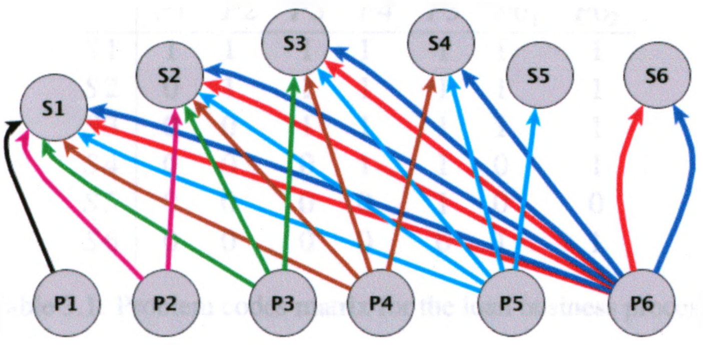


Figure 3.8: Causality graph of the loan business process

each path that ends with *w* create a new path that appends *v* to the path that ends with w.

3.4 Diagnostic Models

This Section shows a diagnostic model used in this research. The coding technique [5] is used to represent our diagnostic models. Each path generated starts from a node *v* and ends at a node *w*. If a problem occurs in node *w* then it is possible that symptoms are detected by each node in the path. Thus each path generated is represented in PCM as a column. We see this with Figure 3.8 and Figure 3.9 and tables 3.1 and 3.2.

By applying the mdfs and pathGenerator algorithms on the loan CD graph, the generated paths are: $((P1), (P1, P2), (P1, P2, P3), (P1, P2, P3, P4), (P1, P2, P3, P4, P5), (P1, P2, P3, P4, P6),$ $(P1, P2, P3, P6))$. After the loan causality graph is determined, PCM is ready to be maintained. From Figure 3.8, we can see that $S5$ can be observed for $P5$ ($\mathrm{PCM}[S5, P5]$) so the $\mathrm{PCM}[5,5]$ is denoted with one. Symptom $S5$ can not be observed for problem $P6$ ($\mathrm{PCM}[S5, P6]$) so $\mathrm{PCM}[5, 6_1]$ has been assigned zero. The PCM matrix for Figure 3.8 is presented in table 3.1. In table 3.1, there are two $P6$ columns ($P6_1, P6_2$) that indicate different patterns of symptoms resulting from problem $P6$. Each pattern corresponds to a path and since there are two paths to the web service corresponding to $P6$ there are two columns.

By apply the mdfs and pathGenerator algorithms on the office CD graph, in Figure 3.9, since $S4$ can be observed for $P4$ the $\mathrm{PCM}[4,4]$ is assigned the value of one. Since symptom

	$P1$	$P2$	$P3$	$P4$	$P5$	$P6_1$	$P6_2$
$S1$	1	1	1	1	1	1	1
$S2$	0	1	1	1	1	1	1
$S3$	0	0	1	1	1	1	1
$S4$	0	0	0	1	1	0	1
$S5$	0	0	0	0	1	0	0
$S6$	0	0	0	0	0	1	1

Table 3.1: Problem codes matrix for the loan business process

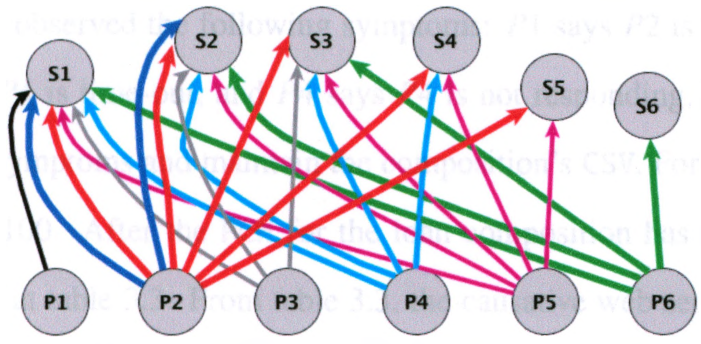


Figure 3.9: Causality graph of the office business process

$S5$ can not be observed for $P6$ PCM[5, 6] has been assigned the value 0. All codes assigned to present the causality relationships in Figure 3.9 are portrayed in table 3.2. In table 3.2, there are two columns representing different patterns that result in symptoms associated with the web service that is associated with problem $P2$.

	$P1$	$P2_1$	$P2_2$	$P3$	$P4$	$P5$	$P6$
$S1$	1	1	1	1	1	1	1
$S2$	0	1	1	1	1	1	1
$S3$	0	0	1	1	1	1	1
$S4$	0	0	1	0	1	1	0
$S5$	0	0	1	0	0	1	0
$S6$	0	0	0	0	0	0	1

Table 3.2: Problem codes matrix for the office business process

Fault diagnosis assumes a vector of symptoms that have been reported. It is assumed that these symptoms are generated by a failure detection component located within a composition. The Hamming distance between the vector and each column is calculated. The Hamming distance the more likely that the column explains what is causing the symptoms.

Assume that the loan business process has been executed. When a set of unpredictable changes of web service behaviour are observed, the loan composition should gather these complaints and pass them to its diagnostic model. These changes are presented as web services' complaints from other web services. An example of these complaints is "$P1$ says $P2$ time out". Such complaints or symptoms will be represented as CSV, where ones denotes that a web service complaints about another web service. Otherwise, CSV will be filled by zeros. If the loan business process observed the following symptoms: $P1$ says $P2$ is time-out, $P2$ says $P3$ is time-out, $P3$ says $P4$ is time-out, and $P4$ says $P4$ is not responding, the diagnostic model should receive these symptoms and maintain the composition's CSV. For this pattern of symptoms, the CSV is 111100. After the PCM for the loan composition has been maintained, the Result list is depicted at table 3.3. From table 3.3, the causative web service for the observed symptoms is ($P4$) because it has the minimum value between its peers.

	$P1$	$P2$	$P3$	$P4$	$P5$	$P6_1$	$P6_2$
$S1$	0	0	0	0	0	0	0
$S2$	1	0	0	0	0	0	0
$S3$	1	1	0	0	0	0	0
$S4$	1	1	1	0	0	1	0
$S5$	0	0	0	0	1	0	0
$S6$	0	0	0	0	0	1	1
$\sum$	3	2	1	**0**	1	2	1

Table 3.3: Result list of the loan business process

Assume that the office business process has been executed. When a set of unpredictable changes of some web services' functionality are observed, the office composition should gather these complaints and pass them to its diagnostic model. If the office composition observed the following symptoms: $P1$ says $P2$ is time-out, $P2$ says $P3$ is time-out, $P3$ says $P4$ is time-out, and $P4$ says $P5$ is time-out, and $P5$ says $P5$ is not responding, the diagnostic model should receive these symptoms and maintain the composition's CSV. For this pattern of symptoms, the CSV is 111110. After After the PCM for the loan composition has been maintained, the Result list is depicted at table 3.4. From table 3.4, the causative web service for the observed symptoms are ($P2_2$ or $P5$) because they have the minimum values between their peers.

	$P1$	$P2_1$	$P2_2$	$P3$	$P4$	$P5$	$P6$
$S1$	0	0	0	0	0	0	0
$S2$	1	0	0	0	0	0	0
$S3$	1	1	0	0	0	0	0
$S4$	1	1	0	1	0	0	1
$S5$	1	1	0	1	1	0	1
$S6$	0	0	0	0	0	0	1
$\sum$	4	3	**0**	2	1	**0**	3

Table 3.4: Result list of the office business process

3.5 Summary

This Chapter covered detailed description of the proposed approach for this research.

Chapter 4

Architecture

This Chapter describes the architecture of the proposed diagnosis module. Section 4.1 provides an overview of the host system for the diagnosis module. Section 4.2 provides an overview of the diagnosis module and the functionality of each component of the module.

4.1 Third Party Management System

This Section describes the context that the diagnostic system is to be used in. We assume that the diagnosis module is a component of a third party policy-based management system[37]. In this thesis we primarily focus on management issues that are concerned with interactions between web services. This management is guided in its decision making by three kinds of policies: *service selection*, *SLA violation* and *recovery* policies [37]. The service selection policy is defined by clients to guide choice of services. The violation policy specifies what constitutes a violation of an SLA. The recovery policy is defined by clients that specifies recovery actions to be taken when the management system detects a SLA violation.

The following case shows how the three policies are used. The clients specify the desired values or range of values for a QoS attribute such as service time. For example, if a client wants a currency rate service that has a service time[1] between 2000 to 4000 milliseconds, the service

[1]Service time is a time taken by the provider to process the service request and generate response [37].

selection policy specifies the desired service time. The policy also specified the number of SLA violations that are to occur before an event is generated. The event represents a failure or symptom. The clients set recovery reactions for each event. If the clients want to set a recovery actions, such as change a service provider,, the recovery policy has to only change providers when the defined violation policy generated events about the selected service.

4.1.1 TPA

A key component in the management system is the third party agent (TPA). The TPA carries out these tasks: (1) allows all clients, providers, and provided services to be registered with it; (2) negotiates SLAs, polices, and keeps track of violated SLAs; (3) generates events to indicate failures and performs recovery actions.

The TPA consists of several components: Registration Gate, Negotiator, Event Generator, Diagnosis Module, and Recovery Agent. An overview of the TPA is presented as Figure 4.1. The Registration Gate is responsible for (1) forwarding a BPEL specification to the BPEL Repository, which stores the BPEL specification for each composition being managed by the TPA. This is one of the inputs for the Diagnosis Module. (2) forwarding relevant information about clients and providers to the Negotiator. The Negotiator is responsible for maintaining an agreement (i.e. SLA) between a client and a service provider if both parties have a match between the former's needs and the latter's specification. These agreements are stored in the Contract Repository. The Event Generator relies on the stored information found in logs storage, such as, information related to service invocations. The Event Generator also "uses SLAs and SLA violation policies to generate events that represent SLA violations ... when the number [SLA violations] exceeds what is specified in the SLA violation policy then an event is generated"[37]. The diagnosis module receives the generated events and uses the BPEL specifications to deliver diagnostic hypotheses. The Recovery Agent is responsible for analysing the diagnosis module's hypotheses and executing reactive actions.

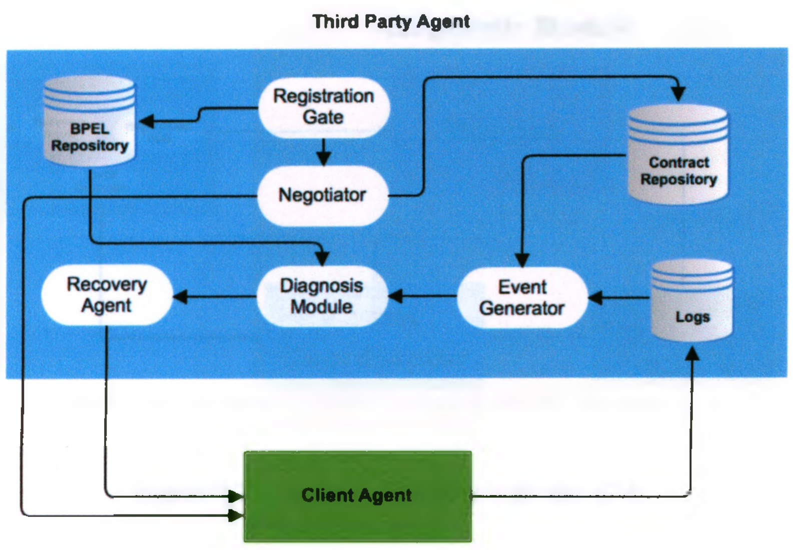


Figure 4.1: TPA with the Client Agent

4.2 Diagnosis Module Overview

Our proposed diagnosis module provides a hypothesis about the source of symptoms observed in a composition. The basic module architecture is presented in Figure 4.2. There are main three components: (1) The Mapper which transforms received BPEL specifications to PCM; (2) The Event Coordinator which transforms the generated events to CSV; (3) The Matcher which is responsible for matching PCM and CSV to deliver a hypothesis. The following Sections describe each component in more detail.

4.2.1 Mapper

An overview of the Mapper is presented in Figure 4.3. The BPMN description of each composition is sent to the Mapper. The Mapper passes the BPMN's description to a component called *Parser*. The Parser receives the BPMN description and prepares a composition graph (i.e. CD graph), which is an input for the *Mdfs* component. A copy of the composition graph is placed in the CD Graphs Storage. The Mdfs component then applies the mdfs algorithm on the composition graph and passes its graph traversal output to the *Path Generator* component.

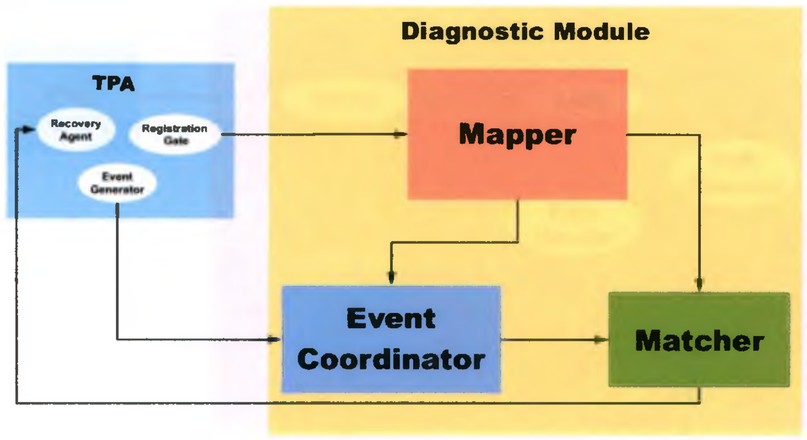


Figure 4.2: Diagnosis module with the TPA

Based on the mdfs algorithm output, the Path Generator component applies the pathGenerator algorithm and delivers paths of the composition graph to *Path Handler* component. The Path Handler uses the generated paths from pathGenerator algorithm and produces all paths from the starting vertex in the composition graph to each adjacent vertex to it until the end of each branch in the composition graph. The new paths represent the causality graph. After generating the causality graph, the Path Handler transforms the causality graph to PCM with respect to the vertices number in the composition graph stored in the CD Graph Storage. Before the end of the mapping stage, PCM is recorded in the Problem Code Matrix Storage. The PCM is now referred to as *composition PCM*. The functionality in the Mapper component is executed once for each composition when the composition is registered.

4.2.2 Event Coordinator

The Event Coordinator receives as input events. The Event Coordinator transforms these events to CSV. Before the transformation, the Event Coordinator requires the number of vertices for the composition graph in order to maintain a vector that is compatible with the size of the PCM generated from the mapping stage. After receiving the number of vertices for the composition graph from the CD Graph Storage, the Event Coordinator creates a vector of zeros. For each element in the vector, the element is replaced by one if its corresponding generated event was

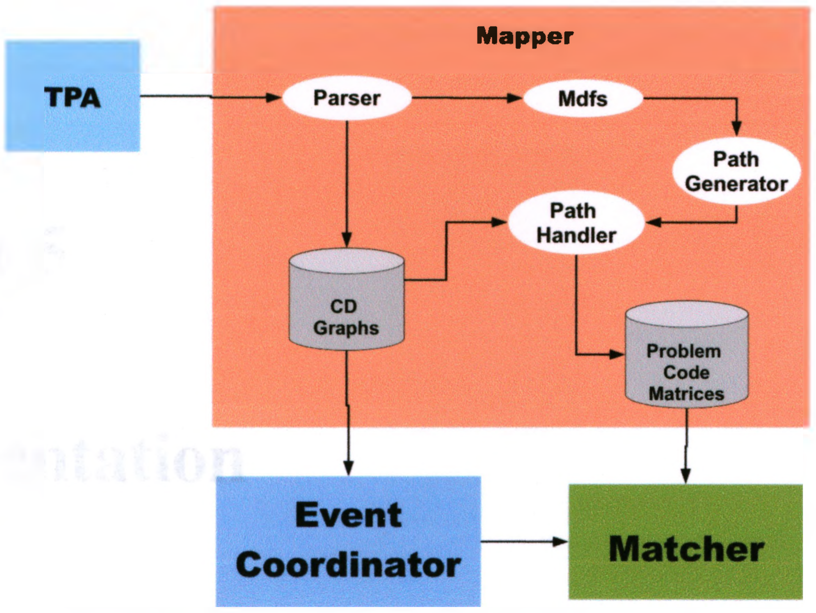


Figure 4.3: Mapper with the diagnosis module components

observed. The CSV is now referred to as *composition CSV*.

4.2.3 Matcher

After the mapping and event coordinating stages are executed, the Matcher component receives both outputs (i.e. PCM, CSV) and applies the Hamming distance procedure for each column in the composition PCM against the composition CSV. The minimum value of the Hamming distance values indicates the source for symptoms occurrences. In the end, the PCM, CSV, and minimum value of the Hamming distance values will be used to determine the faulty services. This information is sent to the Recovery Agent. The Recovery Agent uses recovery policies to determine the appropriate action.

Chapter 5

Implementation

This Chapter gives an overview about the implementation of the diagnosis module. Section 5.1 describes diagnosis module's necessary parts. Section 5.2 describes how the diagnosis module was evaluated.

5.1 Implementation of diagnosis module components

We have implemented the Mapper, the Event Coordinator, and the Matcher. All these components are written in the Java programming language. The implementation details of these components are described in this section. A set of classes were created to perform the diagnosis module's task. Figures 5.1 and 5.2 show this set of classes with a simplified view of a set of properties and methods.

5.1.1 Mapper

The Mapper is an application that has four processing components (Parser, Mdfs, PathGenerator, PathHandler) and two data storages (CD Graph and Problem Code Matrix). Implementation details of these components are described in this section.

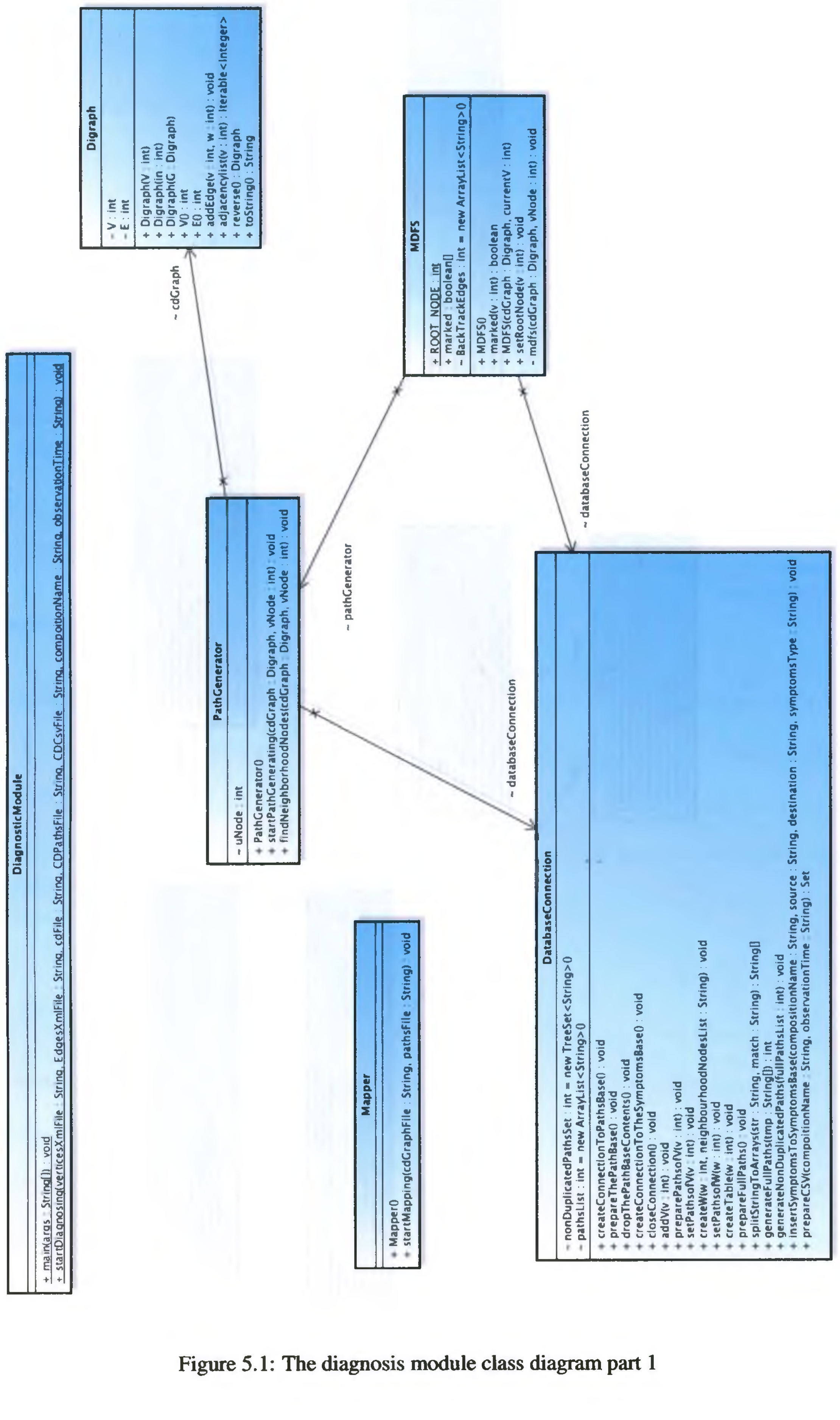

DiagnosticModule
+ main(args : String[]) : void
+ startDiagnosing(verticesXmlFile : String, EdgesXmlFile : String, cdFile : String, CDPathsFile : String, CDCsvFile : String, compoitionName : String, observationTime : String) : void
Digraph
- V : int
- E : int
+ Digraph(V : int)
+ Digraph(in : int)
+ Digraph(G : Digraph)
+ V() : int
+ E() : int
+ addEdge(v : int, w : int) : void
+ adjacencylist(v : int) : Iterable<Integer>
+ reverse() : Digraph
+ toString() : String
~ cdGraph
PathGenerator
~ uNode : int
+ PathGenerator()
+ startPathGenerating(cdGraph : Digraph, vNode : int) : void
+ findNeighborhoodNodes(cdGraph : Digraph, vNode : int) : void
~ pathGenerator
Mapper
+ Mapper()
+ startMapping(cdGraphFile : String, pathsFile : String) : void
MDFS
+ ROOT_NODE : int
+ marked : boolean[]
~ BackTrackEdges : int = new ArrayList<String>()
+ MDFS()
+ marked(v : int) : boolean
+ MDFS(cdGraph : Digraph, currentV : int)
+ setRootNode(v : int) : void
- mdfs(cdGraph : Digraph, vNode : int) : void
~ databaseConnection
~ databaseConnection
DatabaseConnection
- nonDuplicatedPathsSet : int = new TreeSet<String>()
- pathsList : int = new ArrayList<String>()
+ createConnectionToPathsBase() : void
+ prepareThePathBase() : void
+ dropThePathBaseContents() : void
+ createConnectionToTheSymptomsBase() : void
+ closeConnection() : void
+ addV(v : int) : void
+ preparePathsofV(v : int) : void
+ setPathsofV(v : int) : void
+ createW(w : int, neighbourhoodNodesList : String) : void
+ setPathsofW(w : int) : void
+ createTable(w : int) : void
+ prepareFullPaths() : void
+ splitStringToArrays(str : String, match : String) : String[]
+ generateFullPaths(tmp : String[]) : int
+ generateNonDuplicatedPaths(fullPathsList : int) : void
+ insertSymptomsToSymptomsBase(compositionName : String, source : String, destination : String, symptomsType : String) : void
+ prepareCSV(compoitionName : String, observationTime : String) : Set

Figure 5.1: The diagnosis module class diagram part 1

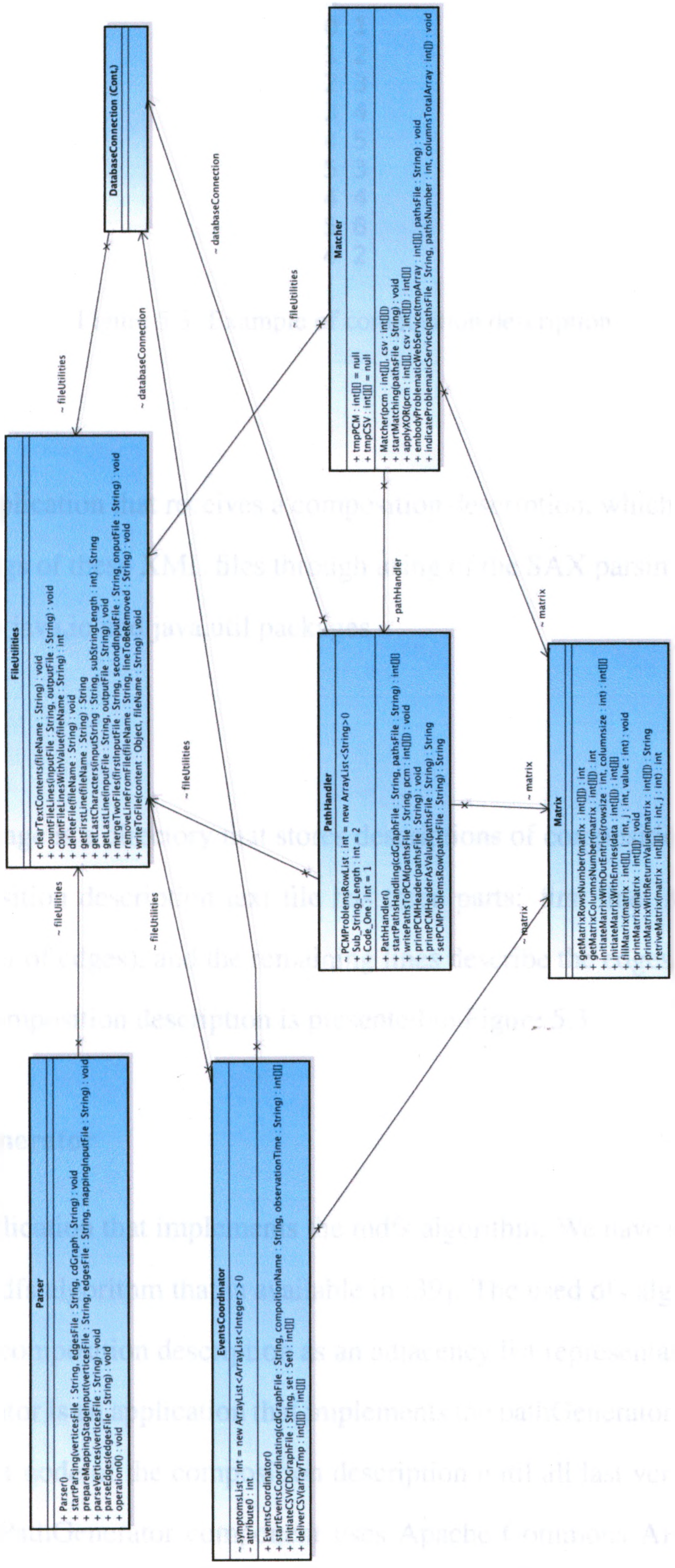

Parser
+ Parser()
+ startParsing(verticesFile : String, edgesFile : String, cdGraph : String) : void
+ prepareMappingStageInput(verticesFile : String, edgesFile : String, mappingInputFile : String) : void
+ parseVertices(verticesFile : String) : void
+ parseEdges(edgesFile : String) : void
+ operation0() : void
FileUtilities
+ clearTextContents(fileName : String) : void
+ countFileLines(inputFile : String, outputFile : String) : void
+ countFileLinesWithValue(fileName : String) : int
+ deleteFile(fileName : String) : void
+ getFirstLine(fileName : String) : String
+ getLastCharacters(inputString : String, subStringLength : int) : String
+ getLastLine(inputFile : String, outputFile : String) : void
+ mergeTwoFiles(firstInputFile : String, secondInputFile : String, onputFile : String) : void
+ removeLineFromFile(fileName : String, lineToBeRemoved : String) : void
+ writeToFile(content : Object, fileName : String) : void
DatabaseConnection (Cont.)
EventsCoordinator
~ symptomsList : int = new ArrayList<ArrayList<Integer>>()
- attribute0 : int
+ EventsCoordinator()
+ startEventsCoordinating(cdGraphFile : String, compoitionName : String, observationTime : String) : int[][]
+ initiateCSV(CDGraphFile : String, set : Set) : int[][]
+ deliverCSV(arrayTmp : int[][]) : int[][]
PathHandler
~ PCMProblemsRowList : int = new ArrayList<String>()
+ Sub_String_Length : int = 2
+ Code_One : int = 1
+ PathHandler()
+ startPathsHandling(cdGraphFile : String, pathsFile : String) : int[][]
+ writePathsToPCM(pathsFile : String, pcm : int[][]) : void
+ printPCMHeader(pathsFile : String) : void
+ printPCMHeaderAsValue(pathsFile : String) : String
+ setPCMProblemsRow(pathsFile : String) : String
Matcher
+ tmpPCM : int[][] = null
+ tmpCSV : int[][] = null
+ Matcher(pcm : int[][], csv : int[][])
+ startMatching(pathsFile : String) : void
+ applyXOR(pcm : int[][], csv : int[][]) : int[][]
+ embodyProblematicWebService(tmpArray : int[][], pathsFile : String) : void
+ indicateProblematicService(pathsFile : String, pathsNumber : int, columnsTotalArray : int[]) : void
Matrix
+ getMatrixRowsNumber(matrix : int[][]) : int
+ getMatrixColumnsNumber(matrix : int[][]) : int
+ initiateMatrixWithOutEntries(rowsize : int, columnsize : int) : int[][]
+ initiateMatrixWithEntries(data : int[][]) : int[][]
+ fillMatrix(matrix : int[][], i : int, j : int, value : int) : void
+ printMatrix(matrix : int[][]) : void
+ printMatrixWithReturnValue(matrix : int[][]) : String
+ retriveMatrix(matrix : int[][], i : int, j : int) : int
~ fileUilities
~ fileUtilities
~ fileUtilities
~ fileUtilities
~ fileUtilities
~ databaseConnection
~ databaseConnection
~ pathHandler
~ matrix
~ matrix
~ matrix

Figure 5.2: The diagnosis module class diagram part 2

```
7
9
0 1
1 2
2 3
3 4
4 5
5 3
4 4
5 6
4 2
```

Figure 5.3: Example of composition description

Parser

The Parser is an application that receives a composition description, which is denoted in XML, and examines all tags of these XML files through using of the SAX parsing methodology [38]. The Parser uses the java.io and java.util packages.

CD Graph

The CD Graph storage is a directory that stores descriptions of compositions presented as text files. Each composition description text file has three parts: first line (number of vertices), second line (number of edges), and the remaining lines describe the edges of the composition. An example of a composition description is presented in Figure 5.3.

Mdfs and PathGenerator

The Mdfs is an application that implements the mdfs algorithm. We have modified an existing implementation of dfs algorithm that is available in [39]. The used dfs algorithm implementation represents the composition description as an adjacency list representation.

The PathGenerator is an application that implements the pathGenerator algorithm by which a path from the root node of the composition description until all last vertices in all branches is delivered. The PathGenerator component uses Apache Commons API [40] and java.util package. An example of maintained paths from the PathGenerator component is presented in

Figure 5.4a.

PathHandler

The PathHandler is an application that maintains all paths needed to represent the causality graph of the composition. Each path represents a new column in a PCM matrix, where each node in a path is replaced with the value of one in the PCM. The PathHandler uses the java.io and java.util packages. An example of maintained paths from the PathHandler component and its PCM are presented in Figure 5.4b and Figure 5.4c, respectively. In Figure 5.4c, the first row represents all possible problematic web services in the composition where each service is titled by a notation (i.e. $Px|y$). The notation has two parts: (1) x is a node that represents a service in the composition description graph; (2) y is an identifier of a path that starts with node x.

Problem Code Matrix

The Problem Code Matrix storage is a directory that stores the PCM for each composition.

5.1.2 Event Coordinator

The Event Coordinator is an application that creates a vector that represents observed events. Each event is replaced by the value of one in the CSV when the event is observed within a composition. The Event Coordinator formulates the vector according to event's source order within the composition. The Event Coordinator uses the Transmorph API [41] and java.util package. An example of observed events and its vector is presented in Figure 5.5a and Figure 5.5b, respectively.

5.1.3 Matcher

The Matcher component finds the minimum Hamming distance values. The Matcher component uses the java.util package.

```
0 1 2 3 4 5 3
0 1 2 3 4 5 6
0 1 2 3 4 4 2
0 1 2 3 4 4
0 1 2 3 4 2
0 1 2 3 4
0 1 2 3
0 1 2
```

(a) Paths from PathGenerator

```
0 1 2 3 4 2
0 1 2 3 4 4 2
0 1 2 3 4 4
0 1 2 3 4 5 3
0 1 2 3 4 5 6
0 1 2 3 4 5
0 1 2 3 4
0 1 2 3
0 1 2
0 1
0
```

(b) Paths from PathHandler

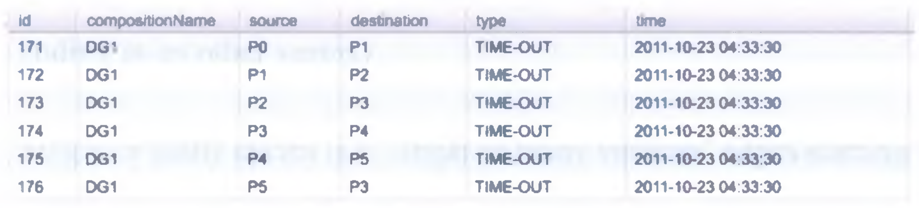
```
P2|10  P2|9  P4|8  P3|7  P6|6  P5|5  P4|4  P3|3  P2|2  P1|1  P0|0
  1      1     1     1     1     1     1     1     1     1     1
  1      1     1     1     1     1     1     1     1     1     0
  1      1     1     1     1     1     1     1     1     0     0
  1      1     1     1     1     1     1     1     0     0     0
  1      1     1     1     1     1     1     0     0     0     0
  0      0     0     1     1     1     0     0     0     0     0
  0      0     0     0     1     0     0     0     0     0     0
```

(c) Example of PCM

Figure 5.4: Example of generated paths

5.1.4 Database

A database is used to store generated paths from the Path Generator and observed events. The database is referred to as *ServiceManager*, which is a MySql database. The diagnosis module uses Mysql Java 5.0.8 connector [42] to interact with the ServiceManager database. The generated paths are stored at a table is referred to as *PathOfNode*, and the observed events are stored in *events* table. Each PathOfNode table represents a node's path. Each PathOfNode table stores node name, successor node name, and successor nodes' paths stream. Each record in a PathOfNode table consists of the following information: start node (v), end node (w), path from w to v. Each record in the events table consists of the following information: name of composition, source node who claimed about the events, node who is claimed about as destination of the event, event type, and time where the composition observed the events.



id	compositionName	source	destination	type	time
171	DG1	P0	P1	TIME-OUT	2011-10-23 04:33:30
172	DG1	P1	P2	TIME-OUT	2011-10-23 04:33:30
173	DG1	P2	P3	TIME-OUT	2011-10-23 04:33:30
174	DG1	P3	P4	TIME-OUT	2011-10-23 04:33:30
175	DG1	P4	P5	TIME-OUT	2011-10-23 04:33:30
176	DG1	P5	P3	TIME-OUT	2011-10-23 04:33:30

(a) Composition observed events

```
1
1
1
1
1
1
0
```

(b) CSV

Figure 5.5: Example of composition observed events and its CSV

5.2 Evaluation

We wanted to test our diagnosis module on composition description graphs to see if the module is able to accurately and correctly determine the source of events.

5.2.1 Hardware

We ran the diagnosis module on a single machine with 2.66 GHz Intel Core 2 Duo processor, Mac OS X 10.6.8 , and eight gigabyte 1.07 GHz memory.

5.2.2 Software

We used Netbeans 7.0.1 IDE to run tests and create or manipulate CSVs. For the transformation from BPMN to the composition description graphs, we used a tool referred to as the BPMN Modeler, which is an extension of eclipse IDE [43]. The BPMN Modeler is responsible for creating a BPMN for a business process and forwarding a BPMN textual description to the Mapper component.

5.2.3 Methodology

We applied our diagnosis module to ten subjects which consists of:

# Single or many joins (i.e. single or many vertices' edges ending in one vertex).

# Single or many splits (i.e. single or many vertices' edges starting from one vertex and ending at an other vertex).

# Single or many cycles (i.e. single or many vertices' edges starting and ending at the same vertex).

# self cycles (i.e. single vertex' edges is starting and ending at the same vertex).

# trees (i.e. single or more vertices are interconnected in a hierarchical manner).

For each performed test, we assumed that one fault could happen for each subject. For each subject we did a test for each web service going down. All evaluation results and composition dependencies graphs are presented in table A.1 and Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8 , A.9, A.10, respectively.

A correct diagnosis was found 100% of the time. In cyclic composition description graphs, the diagnosis module indicates not only the problematic node but also the closest predecessor node to the causative node. The reason is that both the causative node and the predecessor node have the same code in the PCM. Thus, any faults occurring in either these nodes will generate the same events in the composition.

Chapter 6

Conclusion

This thesis relates to the area of web service management and the focus is on mapping the a BPEL specification to a diagnostic module to determine the source of complains within a web service composition. Section 6.1 presents our contributions. Section 6.2 presents possible future work.

6.1 Contributions

By using our diagnosis module the complexity of diagnosis can be hidden from the system administrators by outsourcing this functionality to a third party agent. The proposed diagnosis models enhance the automated diagnosis for a large number of compositions.

Because the codebook technique can not cope with overlapping events or compositions with dynamically changing dependencies, our work generates a new PCM for each change in the structure of a composition. Each new PCM represents a diagnostic knowledge base for the current structure of composition by which the diagnostics' output builds on. The more the diagnostic knowledge base is updated about the composition's possible faulty interactions the more the diagnostic is correct.

Because the diagnosis module does require JDK environment to be executed, we believe that the diagnosis module could be integrated with other systems that require codebook-driven diagnostics capacities. Table B.1 shows the execution time for the examined ten compositions description graphs.

6.2 Future Work

There is a good deal of room for improvement in the diagnosis module. The selection ability of the module for observed events needs to be more consistent in order to pick qualified events. For example, when events are collected for creation of CSVs, some of these events are important and some of them are duplicated or are outdated. These kind of events need to be filtered in order to generate accurate CSVs.

An interval of time needs to be carefully selected such that events generated during that interval are sufficient for analysis. However, an interval that is too long could impact the time it takes to take corrective actions.

The current version of the diagnosis module only uses the the codebook technique. We will enable the module to use several event correlations techniques by which the module will be able to regenerate more efficient diagnostic knowledge bases.

Appendix A

Evaluation Results

The following table shows ten composition description graphs used as subjects for the evaluation of the diagnosis module. The table has 12 columns. The second column shows the name of the composition description graphs. The third and forth columns show the number of vertices and edges of each composition description graphs, respectively. From the fifth column until the 11th column represents aspects about the structure of the composition description graphs. These attributes involve: (1) have single or many cycles; (2) have self cycles; (3) have single or many splits; (4) have single or many joins. The last column shows the time needed to determine the source of observed events, which is *Diagnosis Time*.

No	Composition Description Graph	Vertices Number	Edges Number	Single Cycle	Self Cycle	Many Cycles	Single Split	Many Splits	Single Join	Many Joins	Diagnosis Time
1	Loan CD	6	6					•	•		4 ms
2	Office CD	6	6	•			•		•		3.2 ms
3	CD 1	7	9		•	•					5.8 ms
4	CD 2	6	6	•							2 ms
5	CD 3	10	10					•	•		13.2 ms
6	CD 4	11	12					•		•	9 ms
7	CD 5	16	20					•		•	32.8 ms
8	CD 6	100	114			•		•		•	328.8 ms
9	CD 7	9	11					•		•	9.4 ms
10	CD 8	33	34	•				•	•		32.8 ms

Table A.1: Ten CD graphs specifications

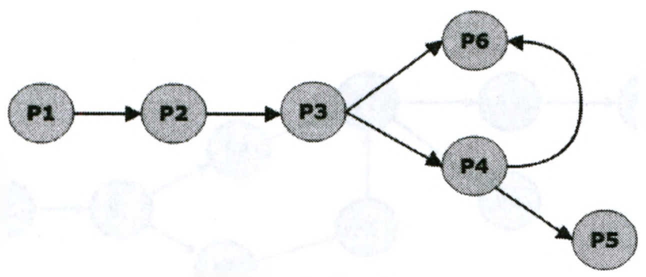


Figure A.1: Loan CD

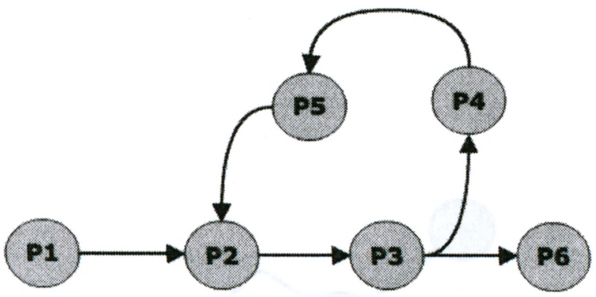


Figure A.2: Office CD

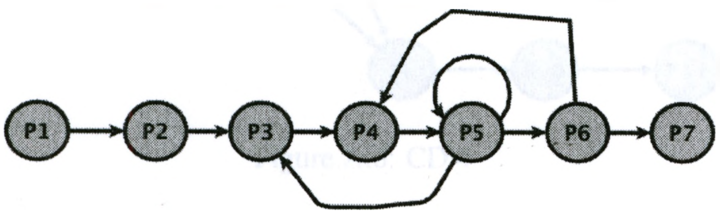


Figure A.3: CD 1

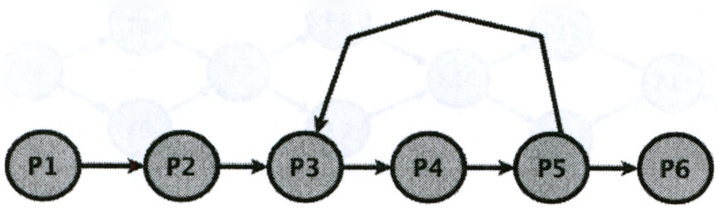


Figure A.4: CD 2

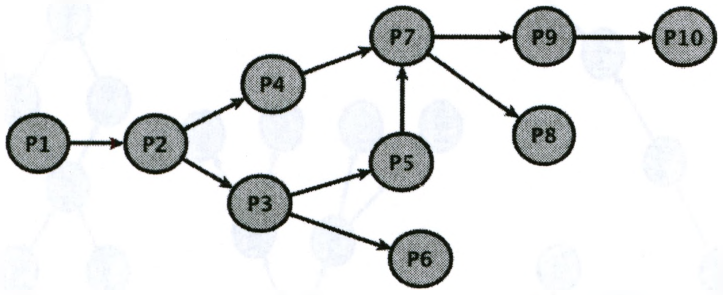


Figure A.5: CD 3

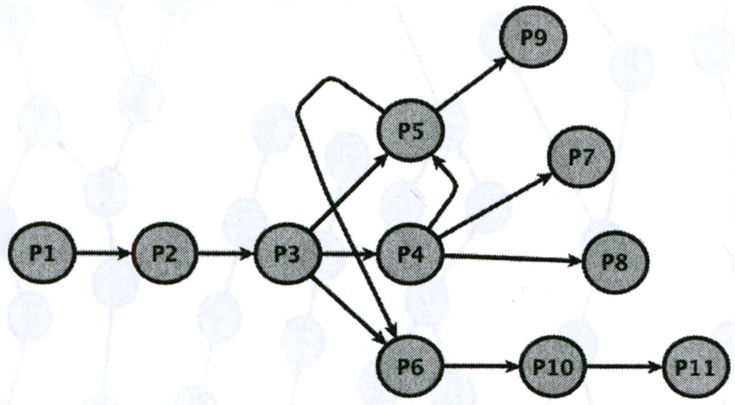


Figure A.6: CD 4

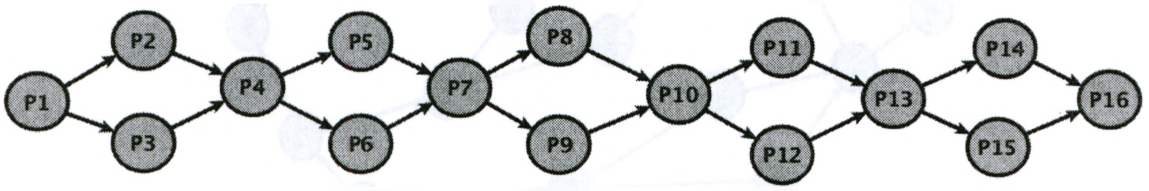


Figure A.7: CD 5

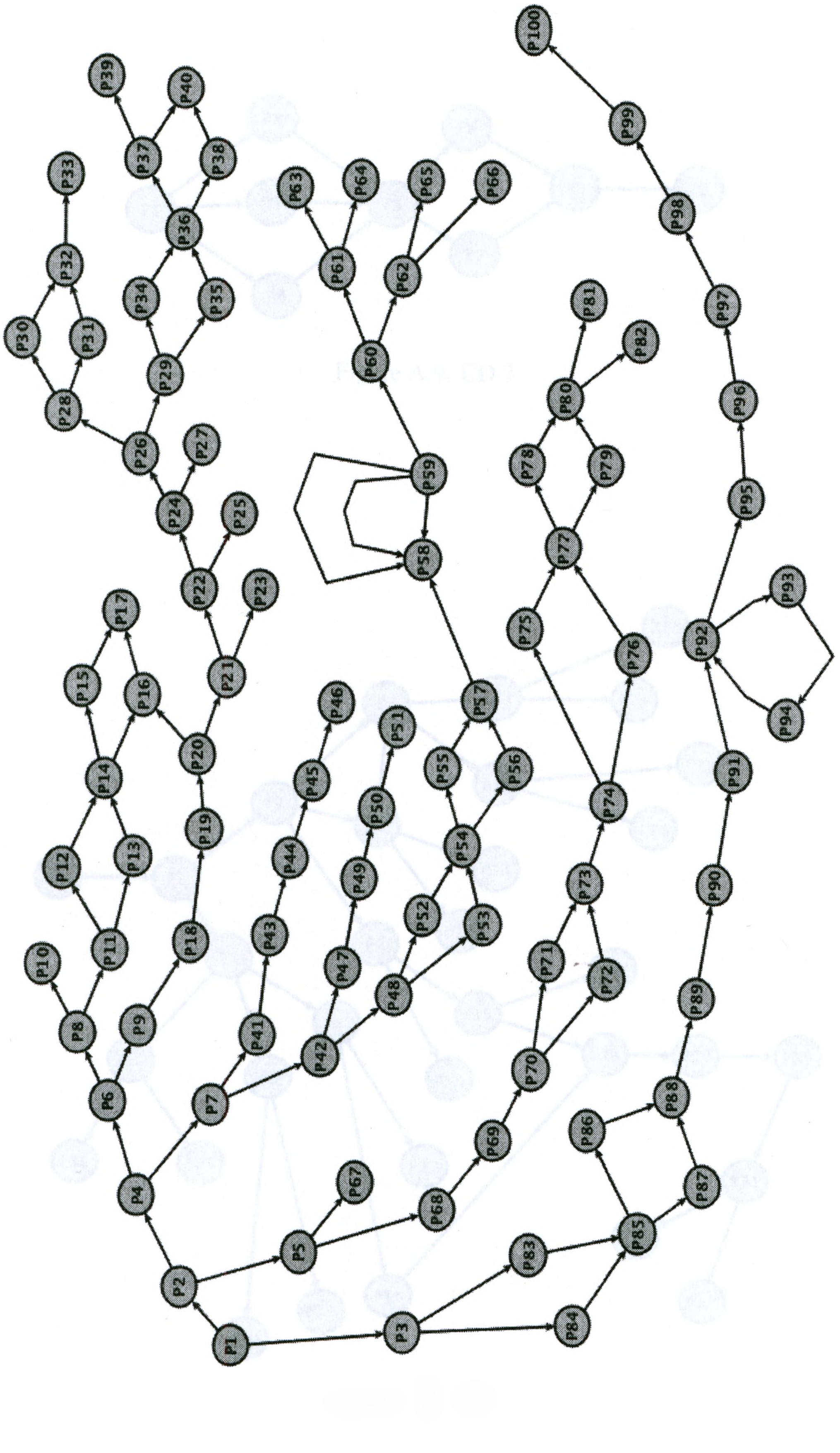


Figure A.8: CD 6

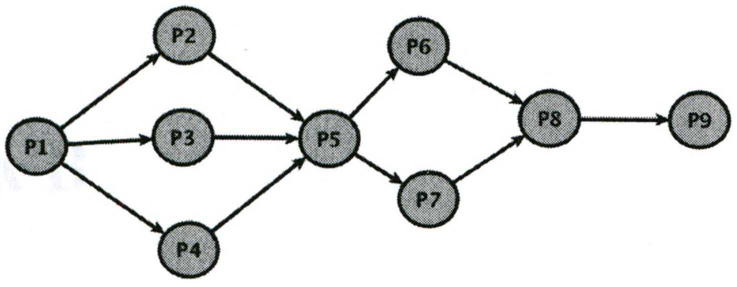


Figure A.9: CD 7

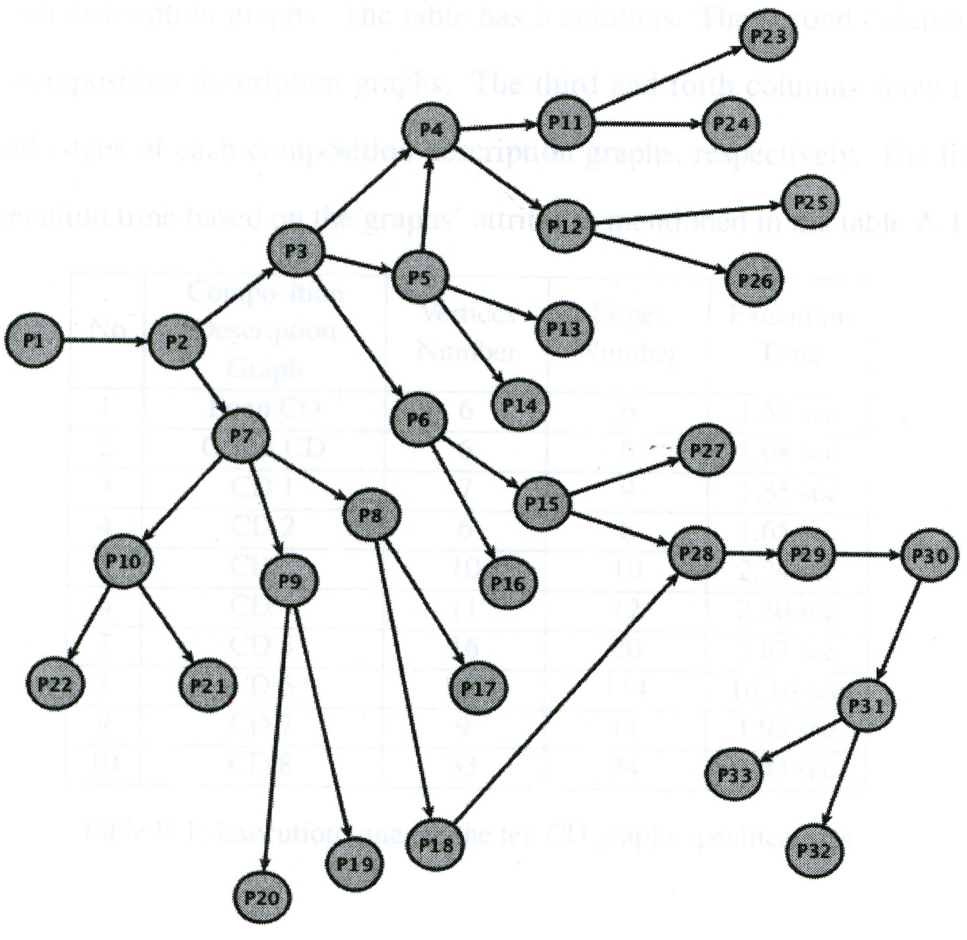


Figure A.10: CD 8

Appendix B

Execution Time

The following table shows the execution time of the diagnosis module needed for each of the ten composition description graphs. The table has 5 columns. The second column shows the name of the composition description graphs. The third and forth columns show the number of vertices and edges of each composition description graphs, respectively. The fifth column shows the execution time based on the graphs' attributes mentioned in the table A.1.

No	Composition Description Graph	Vertices Number	Edges Number	Execution Time
1	Loan CD	6	6	1.57 sec
2	Office CD	6	6	1.68 sec
3	CD 1	7	9	1.85 sec
4	CD 2	6	6	1.65 sec
5	CD 3	10	10	2.24 sec
6	CD 4	11	12	2.30 sec
7	CD 5	16	20	5.87 sec
8	CD 6	100	114	16.16 sec
9	CD 7	9	11	1.93 sec
10	CD 8	33	34	5.43 sec

Table B.1: Execution time for the ten CD graphs specifications

Bibliography

[1] J. jBPM, "WS-BPEL Runtime User Guide." `http://docs.jboss.com/jbpm/bpel/v1.1/userguide/tutorial.hello.html`. Online; accessed 17-Augest-2011.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Springer, 1st edition ed., 2004.

[3] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, *Simple Object Access Protocol (SOAP) 1.1*. World Wide Web Consortium (W3C), May 2000.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium (W3C), March 2001.

[5] M. Steinder and A. S. Sethi, "A survey of fault localization techniques in computer networks," *Science of Computer Programming*, vol. 53, no. 2, pp. 165 – 194, 2004.

[6] Y. Peng and J. A. Reggia, *Abductive inference models for diagnostic problem-solving*. Springer-Verlag New York, Inc., 1990.

[7] M. Keshk, "Executable BPMN: BPMN-2.0 Ontology-Based Native Engine." `http://www.umlowlgen.com/`, 2009. Online; accessed 23-April-2011.

[8] F. H. Zulkernine, P. Martin, and K. Wilson, "A middleware solution to monitoring composite web services-based processes," in *Proceedings of the 2008 IEEE Congress on Services Part II*, (Washington, DC, USA), pp. 149–156, IEEE Computer Society, 2008.

[9] D. Liu and R. Deters, "Management of service-oriented systems," *Service Oriented Computing and Applications*, vol. 2, pp. 51–64, 2008.

[10] O. Levina and V. Stantchev, "Realizing event-driven soa," in *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on*, pp. 37 –42, May 2009.

[11] OASIS, "OASIS Web Services Business Process Execution Language (WSBPEL)." `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`. Online; accessed 17-Oct-2011.

[12] A. Mller and M. I. Schwartzbach, "A WSDL example." `http://www.brics.dk/~amoeller/WWW/webservices/wsdlexample.html`, 2003. Online; accessed 19-Dec-2010.

[13] K. Lee, J. Jeon, W. Lee, S.-H. Jeong, and S.-W. Park, *QoS for Web Services: Requirements and Possible Approaches.* World Wide Web Consortium (W3C) Working Group, November 2003.

[14] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, and F. Casati, “Automated SLA monitoring for web services,” in *Proceedings of the 13th IFIP/IEEE International Workshop on DSOM '02*, pp. 28–41, Springer-Verlag, 2002.

[15] J. Spillner, M. Winkler, S. Reichert, J. Cardoso, and A. Schill, “Distributed contracting and monitoring in the internet of services,” in *Proceedings of the 9th IFIP WG 6.1*, pp. 129–142, Springer-Verlag, 2009.

[16] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, *Grid Resource Allocation Agreement Protocol (GRAAP) WG.* Open Grid Forum, March 2007.

[17] D. Lamanna, J. Skene, and W. Emmerich, “SLang: a language for defining service level agreements,” in *Proceedings of the Ninth IEEE Workshop on Future Trends of FTDCS 2003*, pp. 100 – 106, May.

[18] M. Sloman, “Policy driven management for distributed systems,” *Journal of Network and Systems Management*, pp. 333–360, 1994.

[19] A. Sahai and S. Graupner, *Web Services in the Enterprise: Concepts, Standards, Solutions and Management (Network and Systems Management).* Plenum Publishing Co., 2005.

[20] G. Cabri, “An agent-based architecture for services management,” in *Proceedings of the 18th IEEE International Workshops on WETICE '09*, pp. 19–24, IEEE Computer Society, 2009.

[21] M. Papazoglou and W.-J. van den Heuvel, “Web services management: a survey,” *Internet Computing, IEEE*, vol. 9, no. 6, pp. 58 – 64, 2005.

[22] S. Alam, “Fault management of web services,” master, University of Saskatchewan, 2009.

[23] A. Garza, J. Serrano, R. Carot, and J. Valdez, “Modeling and simulation by petri networks of a fault tolerant agent node,” in *Analysis and Design of Intelligent Systems using Soft Computing Techniques* (P. Melin, O. Castillo, E. Ramrez, J. Kacprzyk, and W. Pedrycz, eds.), vol. 41 of *Advances in Soft Computing*, pp. 707–716, Springer Berlin / Heidelberg, 2007.

[24] A. Hanemann, *Automated IT Service Fault Diagnosis Based on Event Correlation Techniques.* PhD thesis, LMU Mnchen: Faculty of Mathematics, Computer Science and Statistics, 2007.

[25] C. Angeli and A. Chatzinikolaou, “Online Fault Detection Techniques for Technical Systems: A survey,” *International Journal of Computer Science and Applications*, vol. 1, pp. 51–64, 2004.

[26] I. Hwang, S. Kim, Y. Kim, and Seah, “A survey of fault detection, isolation, and reconfiguration methods,” *Control Systems Technology, IEEE Transactions on*, vol. 18, pp. 636–653, may 2010.

[27] M. Tiffany, “A Survey of Event Correlation Techniques and Related Topics.” `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5339`, 2002. Online; accessed 19-Dec-2010.

[28] C. Caldwell, “Graph Theory Glossary.” `http://www.utm.edu/departments/math/graph/glossary.html#b`, 1995. Online; accessed 05-June-2011.

[29] Wikipedia, “Hamming distance.” `http://en.wikipedia.org/wiki/Hamming_distance`, January 2011. Online; accessed 21-Jan-2011.

[30] M. Tighe, “Diagnosis in Policy-Based Autonomic Management,” master, University of Western Ontario, 2009.

[31] Y. Peng and J. A. Reggia, “Plausibility of Diagnostic Hypotheses: The Nature of Simplicity,” AAAI-86, pp. 140–145, 1986.

[32] R. M. Bahati, M. A. Bauer, E. M. Vieira, and O. K. Baek, “Using policies to drive autonomic management,” in *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, WOWMOM ’06, pp. 475–479, IEEE Computer Society, 2006.

[33] R. M. Bahati, M. A. Bauer, and E. M. Vieira, “Policy-driven autonomic management of multi-component systems,” in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON ’07, pp. 137–151, ACM, 2007.

[34] K. S. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea, “Service-oriented computing - icsoc 2007 workshops,” ch. A Fault Taxonomy for Web Service Composition, pp. 363–375, Springer-Verlag, 2009.

[35] S. Bruning, S. Weissleder, and M. Malek, “A fault taxonomy for service-oriented architecture,” in *High Assurance Systems Engineering Symposium, 2007. HASE ’07. 10th IEEE*, pp. 367 –368, 2007.

[36] H. Endert, B. Hirsch, T. Küster, and S. Albayrak, “Towards a mapping from bpmn to agents,” AAMAS’07/SOCASE’07, pp. 92–106, 2007.

[37] M. S. Hasan, “Policy Based Third Party Web Service Management,” master, University of Western Ontario, 2011.

[38] Oracle, “SAX Parser.” `http://download.oracle.com/javase/1.4.2/docs/api/javax/xml/parsers/SAXParser.html`. Online; accessed 17-Oct-2011.

[39] R. Sedgewick and K. Wayne, “DirectedDFS.” `http://algs4.cs.princeton.edu/42directed/DirectedDFS.java.html`. Online; accessed 17-Oct-2011.

[40] ApacheFoundation, “CommonsCollections.” `http://commons.apache.org/collections/`. Online; accessed 17-Oct-2011.

[41] Transmorph, “Transmorph Project Wiki.” `http://transmorph.sourceforge.net/wiki/index.php/Main_Page`. Online; accessed 17-Oct-2011.

[42] MySqlConnector, "ConnectorJ." http://dev.mysql.com/downloads/connector/j/5.0.html. Online; accessed 17-Oct-2011.

[43] Eclipse, "BPMN Modeler." http://eclipse.org/bpmn/. Online; accessed 17-Oct-2011.