Team-oriented Model for Composite Web Services Failure Recovery

Yuan Yi, Chok

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 2005
Abstract

As far as Web services consumers are concerned, functionality is the most important attribute that determines the usefulness of a Web service. However, recent research shows that non-functional attributes such as robustness and availability are equally as important. Since without availability, even the service with the best functionality would serve no purpose to its potential clients. There are many factors that may cause these services to be unavailable and they can be grouped into three different categories: hardware failures, software failures, and human failures.

Failures are inevitable in Web services, at both atomic service level and composite service level. A composite Web service is essentially a combination of smaller services to provide services that cannot be achieved by any single service. Of which, some of these smaller services can be composite services as well. As a result, a composite service is more susceptible to failures than an atomic service. This is due to its dependency on other services, which are external modules to the composite service. As a composite service only knows the external bindings of its constituent services and not their internal structures, combining the constituent services together, in the worst cases, may aggregate their failure occurrences. In this research, we are not interested in monitoring and detecting these failures like most researches do. Instead, we are more interested in how to recover from a composite service failure.

Web services are intrinsically built in a redundant manner as profitable services often encourage competition from various companies. Redundancy and replication of Web services offers reliability since redundant services can be used as replacements or substitutes for similar services that failed in a service composition. Although there are ways to make atomic Web services robust at the service providers’ end, we want to provide failure recovery at a composition level as a service. In this research, we studied two main approaches towards composite service recovery, one is built on traditional distributed system failure recovery to maintain the ACID properties (i.e. backward failure recovery) and the other is based on exception handling in coordinated atomic actions (i.e. forward failure recovery). Both are proposed for consideration as standards. As service-oriented and agent-oriented research are often run in parallel, in this project, we evaluated a team-oriented agent language - JACK Teams, which is renowned for the ability to coordinate modular autonomous components in face of failure. This research
examined how teamwork can be used to provide both forward and backward error recovery as well as efficient failure recovery in composite services.

**Keywords:** Web service composition, team-oriented, failure recovery

**CR Categories:** I.2.11 Distributed Artificial Intelligence.
I wish to acknowledge a group of people, without whom this dissertation would not have been possible.

First and foremost, I must thank my supervisor, Dr. Wei Liu, for her help and patience with me. Her experience and expertise guide me through the entire honours year. Her spontaneity and creativity during our discussion gave me great inspiration for my dissertation writings. I could not have done a better job without her.

I must also thank my mum for providing me with strong emotional support. Even though she cannot be with me physically here in Perth, her daily phone calls pushed me on.

Finally, I wish to thank my fellow course-mates in Bachelor of Computer Science, UWA. Particularly, Gabriel Foo, Sern Wei Yee, and Nikki Tan for accompanying me and giving me words of encouragement throughout our courses.

Cheers, guys.
## Contents

Abstract iii  
Acknowledgements iv  

1 Introduction 1  
1.1 Background 1  
1.2 Why failures in Web services are inevitable? 2  
1.2.1 Hardware failures 2  
1.2.2 Software failures 3  
1.2.3 Human failures 3  
1.3 Recoverable and irrecoverable failures 4  
1.4 Why does service composition need to take care of failure recovery? 4  
1.5 What are the existing error recovery techniques? 5  
1.6 How JACK\textsuperscript{TM} Teams fit into the picture? 6  
1.7 Research aims 6  
1.8 Dissertation structure 7  

2 Literature Review 9  
2.1 Lifecycle of Web services 9  
2.1.1 Atomic Web services lifecycle 9  
2.1.2 Composite Web services lifecycle 10  
2.1.3 Creation of Web services 12  
2.1.4 Description of atomic services 14  
2.1.5 Description of composite services 15  
2.1.6 Publishing and discovery of Web services 16  
2.1.7 Execution of Web services 17  
2.1.8 Why JACK Teams is better in executing Web services? 17
2.2 Existing error recovery techniques ........................................... 18
  2.2.1 Backward error recovery based on transactional attitudes .. 18
  2.2.2 Forward error recovery based on Coordinated Atomic Actions 23

3 Methodology ................................................................. 27
  3.1 Why backward and forward error recovery alone is unsuitable? . 27
  3.2 Hierarchical team architecture technologies in agent community . 29
  3.3 Concepts of JACK Teams and how it meets the needs of composite
      service failure recovery? ............................................... 30
    3.3.1 Team Entity ....................................................... 30
    3.3.2 Team ............................................................. 30
    3.3.3 Sub-team ........................................................ 31
    3.3.4 Role ............................................................. 31
    3.3.5 Teamplan ........................................................ 31
    3.3.6 Agent-oriented concepts ........................................ 31
    3.3.7 How JACK Teams satisfies the required properties for com-
           posite service failure recovery? ............................. 32
  3.4 Providing failure recovery for service composition in JACK Teams 33
    3.4.1 Conceptual mappings between JACK Teams and service
           composition ...................................................... 34
    3.4.2 Persistency at individual and team level .................... 35
    3.4.3 Making use of redundancy .................................... 37
  3.5 A new economical failure recovery proposal .......................... 38

4 Implementation ............................................................. 41
  4.1 Creating and deploying component services .......................... 41
  4.2 Building client stubs from WSDL ..................................... 42
  4.3 Developing JACK Teams codes ....................................... 43
  4.4 Implementing the economical failure recovery ...................... 45
  4.5 Case study ........................................................... 46
    4.5.1 Invocation of Web services ................................. 47
    4.5.2 Persistency at individual level and the containing team level 47
B.4.2 Role ............................................. 67
B.4.3 Plans ......................................... 67
B.4.4 Team .......................................... 68

B.5 Composite service codes .......................... 68
   B.5.1 Event ....................................... 68
   B.5.2 Teamplan .................................... 68
   B.5.3 Team ........................................ 72

B.6 Other codes ...................................... 73
   B.6.1 Definition file ............................... 73
   B.6.2 Main Java program ......................... 73
   B.6.3 Data structure ............................... 74
List of Tables

3.1 Properties required for composite service failure recovery . . . . . . 28
4.1 WSDL2Java server-side bindings conversions [2] . . . . . . . . . . . 42
4.2 WSDL2Java client-side bindings conversions [2] . . . . . . . . . . . 43
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Supporting technologies that are currently available for Web services.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>A typical lifecycle of an atomic Web service.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>A composite Web service created by a service consumer.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>A composite Web service created by a service registry.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>A comparison in team formation between JACK Teams and Team-core.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Interactions between team members in a teamplan.</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of JACK Teams code generation.</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Expected results for each test case.</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Print screen of actual result for test cases (1) and (2).</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Print screen (a) of actual result for test cases (3), (4) and (5).</td>
<td>52</td>
</tr>
<tr>
<td>5.4</td>
<td>Print screen (b) of actual result for test cases (3), (4), and (5).</td>
<td>53</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Background

Generally, an atomic Web service is an application hosted on a company’s Web server to provide a form of service for its clients via the Internet. It is similar to a Web site except that a Web service is meant to be interpreted by machines rather than by human beings. The underlying implementation of Web services involves a collection of protocols and standards to exchange data between applications or systems. Simple Object Access Protocol (SOAP) allows Web services to have a common language to exchange messages with each other. Web Services Description Language (WSDL) provides interface descriptions of the application so that client programs can figure out how to communicate with these services. Web services enable software applications written in various programming languages and running on various platforms to exchange data. The communication, message exchange, and application interoperability provide opportunities to develop new applications and integrate existing diverse applications within a company or beyond company boundaries.

Even though a vast number of Web services are available nowadays, few actually solve the users’ problems. It is often required to combine smaller atomic services to provide services that cannot be achieved by any single service alone. This is called service composition and the service that composes other services is called a composite service. Imagine moving into a new apartment, we would need to subscribe to utility, telephone, gas, and Internet services. Despite these services being available individually, it is desirable to achieve such general tasks by an integration of these individual services. This will save the user lots of time and hassle than to approach each service provider at a time. One way to overcome this difficulty is to compose relevant services together into composite services.

At present, Web services standards have evolved to provide us with description, a platform-independent communication mechanism for Web services, and
some specifications to specify service composition. The Web services platform now offers a distributed computing environment with a large pool of resources and business processes for enterprises to tap onto. New, larger services can be delivered by assembling existing services through service composition without compromising the autonomy of each individual Web service. In fact, composite service is the main drive for large scale service-oriented environment adoption due to its potential of leveraging existing business data and information sharing to business process integration.

1.2 Why failures in Web services are inevitable?

Availability is the most important metric for modern computer systems [7]. Even though Web services are usually chosen for their functionalities, they would be of no use to their consumers if these services are unavailable for most of the time. Especially in the Web services environment where interactions with all services may have to pass through unreliable networks, which are bound to cause communication failures to a certain degree. Furthermore, data access and the functionalities of these services are exposed to consumers through a cluster of interconnected systems. This cluster of systems may include databases, application servers, middleware, and Web servers constructed from multi-vendor Commercial Off The Shelves (COTS) hardware and software, chances of these systems being incompatible with each other do exist. Hence, companies should always look to provide high availability solutions for their services in order to attract more consumers.

Higher availability for Web services may be achievable by these companies. However, whether these companies are willing to compromise other aspects of their services for higher availability or not is up to them to decide. Nevertheless, companies should also be realistic enough to recognize that unanticipated failures are unpreventable. In this section, we discussed what factors may cause even the most unlikely failures to occur.

1.2.1 Hardware failures

One common approach to enhance system robustness is to use more sophisticated hardware. Some examples of such solutions are the usage of fault tolerant or redundant hardware for hosting the services. However, companies offering Web services often work on limited profit margin and they may not be able to afford these expensive fault tolerant hardware. Furthermore, one can go for hardware
with similar functionalities, but more failure-prone at a much cheaper price. Such options make it even tougher for companies to decide. As a result, some of these companies would prefer cost-cutting measures and choose cheaper hardware to support their services. As a result, the reliability of their Web services may be reduced due to occasional hardware failures.

1.2.2 Software failures

Another approach to improve robustness is for companies to conduct extensive software testing before releasing the Web services. However, the world of Web services emphasises on the delivery of new functionalities. Companies competing with their competitors often want to deliver their services first in order to take a bigger share of the Web services market. As a result, the software applications are often rushed to completion, overlooking important aspects of robust system design techniques and cramping the software testing process. Furthermore, some companies may adopt cost-cutting measures and employ inexperienced software developers to build their Web services. These aforementioned factors contribute to the inevitability of software failures, which ultimately compromise the reliability of the Web services.

1.2.3 Human failures

All hardware experience wear and tear, while most software experience changes in their requirements throughout their development process. Human beings are thus relied upon to repair, replace and upgrade hardware, or fix and extend software. However, we all know that humans make mistakes. This is a fact of life. Even in straightforward task like reading numerical digits off a screen, the probability of human mistakes is nonzero [6]. Not to mention that under stressful situations, human error rates tend to rise between ten to a hundred percent [7]. Hence, we can hardly expect the complex and cumbersome maintenance and repair procedures to be error-free. A traditional high-availability solution is to deploy certified, trained technicians as system administrators. However, even highly-trained personnel may make mistakes and these inevitable mistakes will eventually snowballed to cause Web services to fail.

Web services operate in an environment that is both dynamic and open. In this dynamic and competitive environment, Web services come and go without any prior warning. However, some services may generate little or no profit at all for their companies. As a result, these companies may terminate those non-profitable services and concentrate their resources on more lucrative services.
The open environment of Web services meant that besides the implementation of the Web services, the systems supporting them are dynamic and maybe unreliable as well. Hence concurrent usage of certain services by various consumers may sometimes cause the less capable supporting systems to be overloaded, or previously available services may cease to operate without warning.

1.3 Recoverable and irrecoverable failures

From the service consumers’ perspectives, the aforementioned failures in Section 1.3 are deemed as irrecoverable failures. This is due to the fact that these failures occur at the service providers’ ends and there is nothing that the consumers could do to recover from them, except to choose an alternative service that can perform the failed task. Besides using replication or redundancy to cope with these irrecoverable failures, another part of our research is to look at failures that can be recoverable by the consumers’ interventions. A scenario of such recoverable failures could be when a consumer tries to book a hotel room for a certain period of time, which unfortunately the specific hotel accommodation service has all its rooms booked for the requested duration. This room booking operation is then considered to have failed. However, the consumer can still recover from this failure by changing the dates of the booking without having to select alternative hotel room booking service. These recoverable failures are usually dealt with by altering the invocation parameters.

1.4 Why does service composition need to take care of failure recovery?

From the previous sections, we have discussed that Web services failures are inevitable. Hence more attention should be given to develop ways to recover from these failures. Although there have been ongoing research efforts to achieve high reliability for Web services [1], due to the autonomy of these services, we must realise that not all companies will provide failure recovery procedures for their services. Assuming that the failure of any component services will cause the entire composite service to fail, a composite service is only as reliable as its least reliable constituent service. However, it was also mentioned earlier that profitable services encourage competition from various companies. Therefore, it is very likely that redundant services will be available. Just like hundreds or more Web pages with similar information may exist on today’s Web.
Redundancy and replication provide reliability. With redundant services available, failure recovery is made possible as the failed component services may be replaced by other alternatives of the same functionality. Since not all companies would employ the failure recovery methods and that unreliable services may be involved in composite services, it is important that service composition must take care of error recovery. Nonetheless, changing any constituent service in a composite service may lead to some decision-making process; the service composition has to decide whether to dump all previous interactions between the constituent services or to roll back to a previous correct state but retaining all unaffected interactions. This problem leads to our research in making Web service composition more robust and reliable.

1.5 What are the existing error recovery techniques?

Error recovery techniques are mainly used for the development of reliable distributed systems. Over the last couple of years, developing fault tolerant mechanisms for composite Web services has been an active area of research [10]. Error recovery is part of three sequential tasks required to correct an error or a system failure [12]. Backward error recovery and forward error recovery techniques are two main techniques that can be used to ensure the robustness of Web service composition.

**Backward error recovery** is an attempt to restore the system state after an error has occurred. It is based on rolling system components back to the previous correct state. Backward error recovery can be considered as a function to go back in time as it basically removes all actions after the previous correct consistent state. In **forward error recovery**, the application has to execute unexpected tasks to recover from an error and these unexpected tasks involve transforming system components into any correct state. Forward error recovery is usually application-specific and is realised using exception handling mechanism, whereas the backward error recovery uses diversely-implemented software and is usually realised by distributed transactions logs.

Composite Web services are in themselves a special type of distributed systems, because their processing is mostly not done locally but distributed to various system components. The most beneficial way in building such systems and providing their fault tolerance is by measuring the fault tolerance of the intended systems with system structuring units [10]. Distributed transactions and atomic actions are two well known structuring units used to provide failure recovery for Web services. Distributed transactions use backward error recovery to satisfy completely or partially Atomicity, Consistency, Isolation, and Durability (ACID)
properties, whereas atomic actions depend on all involved action participants to coordinate in handling the action exceptions.

1.6 How JACK\textsuperscript{TM} Teams fit into the picture?

Service composition can be viewed as a loose form of teamwork [19]. The structure of a team is generally made up of different individuals. For example, a soccer team can have eleven members plus reserves where each has his/her own beliefs and attributes. However, all members share a common goal, which is to execute all required tasks according to a strategy plan in order to win matches. These teams usually have a team manager to coordinate its activities such as selecting the right strategies for different matches and switching plans when a previous one fails. The reserves in the team act as a form of backup to replace any members that may be injured during a match. We can view a match as an user’s service request, the eleven team members as component services involved in a composite service, and the reserve team members as replacement services in case of any failure in the component services during their execution. Therefore by incorporating composite Web services as teams, the failure recovery mechanism of teamwork can also be provided for composite services. This is a highly desirable property, particularly, as Web services operate in an open environment that has a high failure rate.

JACK\textsuperscript{TM} Teams is one such framework that adopts the team-oriented approach. In this research, we looked at how this agent-oriented team architecture can be used to implement failure recovery for composite Web services and how it can be used to coordinate the execution of composite services.

1.7 Research aims

The main aim of this research is to leverage on the default failure recovery mechanism in an existing Beliefs, Desires, and Intentions (BDI) agent team architecture, named JACK Teams, to provide failure recovery for composite Web services. We suggest that Web service composition can be regarded as an act of cooperation from various business enterprise, some of which may be independent and have no business relations with others. Although these organisations do not necessarily know of their involvement in such cooperation, we can be certain that they are willing participants since advertising their services had indicated a keenness for consumers to make use of them [19]. We also suggest that a composite service can be considered as a set of steps that define the selection of component
services and its coordination with the chosen services during execution. This set of steps can be seen as a teamplan to be executed by a selected group of business organisations. However, as the results of certain component services may not be critical to the outcome of the composite services [18], there may exist more than one teamplan for each composite service.

This research explores the existing error recovery techniques namely, backward error recovery and forward error recovery, to gather suitable properties for composite service failure recovery. This set of properties can be regarded as a set of guidelines for any future recovery methods. In addition to providing default failure recovery mechanism for composite services, this research also looks at extending JACK Teams’ failure recovery mechanism by incorporating a data structure to log the execution of steps in a teamplan. This is to prevent successfully-executed steps from being re-executed when a teamplan fails. The outcome of this research is then to use JACK Teams to represent and execute composite Web services reliably, while improving the efficiency of JACK Teams’ default failure recovery mechanism.

We designed and implemented a system that translates Web services into JACK Teams concepts and allows service consumers to specify teamplans for composite Web services. The composite service is executed using using JACK execution engine, which includes invoking of its component services. We also implemented a failure recovery algorithm in the teamplans of a JACK Teams program using the case study of a travel agency composite Web service to provide experimental results of the improved, economical failure recovery in the presence of failures. The research has convincingly demonstrated through implementation that BDI agent platforms, especially, team-oriented framework can be used to enhance the reliability of composite services.

1.8 Dissertation structure

Taking a lifecycle perspective, Chapter 2 reviews the XML-based standards that support Web services. This is followed by a discussion on how the existing error recovery techniques can be used to provide failure recovery for service composition. Chapter 3 discusses the necessary properties to support composite service failure recovery as specified by the existing error recovery standards. This is followed by an investigation on why existing error recovery techniques may not be as suitable as JACK Teams when it comes to providing failure recovery for composite services. Chapter 3 also discuss about the three different levels of default failure recovery mechanism in JACK Teams and propose a new economical failure recovery for service composition. Chapter 4 describes the four main
parts in the implementation phase. This is followed by a description on the im-
plemented JACK Teams representation of a composite Web service and how the
experiments are being conducted to demonstrate the effectiveness of the proposed
failure recovery algorithm. We discuss the future work that may take place in
Chapter 5 and conclude our research in Chapter 6.
CHAPTER 2

Literature Review

This chapter provides an overview on Web services, specifically their technological support, using a lifecycle approach. This is followed by a discussion on how the error recovery techniques are currently employed to provide reliability for Web services. Section 2.1.1 discusses the different phases that an atomic Web service will have to go through before being available to its potential users. Section 2.1.2 illustrates the lifecycle of a composite Web service created either by the service consumer or the service registry. The rest of Section 2.1 discusses the characteristics of the available supporting technologies for Web services and how a team-oriented modeling framework like JACK Teams is suitable for robust execution of a composite service. Lastly, Section 2.2.2 discusses the two main error recovery techniques that are commonly employed as means to provide reliability and fault tolerance for distributed systems.

2.1 Lifecycle of Web services

In this section, we presented the various supporting technologies available for different phases of a typical Web service’s lifecycle. From the lifecycle perspective, we can clearly separate the large collection of available technological support into the different phases that they are used in. This covers the available technologies of both atomic and composite Web services for their creation, description, publishing and discovery, and execution phases. The detailed information of each phase in Figure 2.1 is provided in the subsections.

2.1.1 Atomic Web services lifecycle

A typical atomic Web service goes through a series of phases during its lifecycle and this lifecycle is being illustrated in Figure 2.2. The hexagons in Figure 2.2 represent the actors, the rectangles symbolize the processes that the service has
An atomic service’s lifecycle begins when a service provider (Provider) decides to create a Web application to serve its consumers. During the creation process, the Web service to be created is described and published in a service registry (Registry) to advertise to its potential consumers. A service consumer (Consumer) at any point of time sends a query to a Registry to look for a specific type of service. The Registry looks up its repository to locate services that match the description of the query. If the right services are found, the Registry returns the results to the Consumer, whom will use the concrete binding information returned by the result to bind to the Provider. Once the binding between the atomic service and the Consumer is done, the Consumer can then proceed to execute the requested service.

### 2.1.2 Composite Web services lifecycle

The lifecycle of a composite Web service is more complicated than that of an atomic service as the composition plan can be created either by a Consumer or a Registry. Each composition plan is made up of control structures and the service types. The control structure controls when sub-services are to be executed, whereas the service types correspond to the roles that a particular sub-service is to play in the composition plan. Before the composition plan gets executed, the roles of each sub-service will be fulfilled by selecting one of the services returned by the Registry that match the desired service type.

Figure 2.3 illustrates a sequence of interactions that are required to take place for a Consumer to create a composition plan. The sequence diagram begins with the individual Providers registering their services with a Registry. A Consumer
then sends a query to the Registry looking for services that matches the desired service type. The Registry would inform the Consumer if no such service was found in its repository. The Consumer would then break down its desired service into several sub-services and sends new queries to the Registry again. If services registered under the Registry’s repository match these queries, the resulting list of services will be returned to the Consumer. The Consumer then selects one of the services and proceeds to bind and execute the sub-services. For simplicity sake, the composition plans in both Figure 2.3 and Figure 2.4 demonstrated the sequence of order in which these sub-services may be executed.

Figure 2.4 demonstrates how a Registry creates a composite service. Similarly, Figure 2.3 begins with Providers registering their services with a Registry. This is followed by a Consumer sending a query to the Registry looking for services that match a specific service type. If the Registry does not record such a service, it would break the service type into several sub-service types and search again within its repository for services that match these sub-service types. Once the composite service is created, the Registry sends the composite service back to the Consumer. The Consumer then binds to the services that the Registry had composed and executes them according to the control structure of the composite service.

The composition plans created by a Consumer are more flexible as a Consumer is able to choose from the group of services returned by a Registry that matches the service types in its queries. However, this would require more mes-
sages to be exchanged between the Consumer and the Registry as the consumer has no overview of the available services. On the other hand, the composition plans created by the Registry require less message exchanges as the registry knew what services are available or registered in its repository. However, the composition plan will not enjoy the same flexibility as the consumer approach. Current Registries such as UDDI or OWL-S do not support the creation of composite service on behalf of the Consumer. Mainly, Registries can not execute the selected services for the Consumer due to user legal responsibilities and the fact that the Consumers and not the Registries should foot the bills for using the services.

2.1.3 Creation of Web services

The growth of Internet technologies has seen the Web evolved from information publication to complex transaction processing. Even till today, it has continued to evolve. Recently, Web services are introduced as an emerging, new model in driving digital economy among distributed and heterogeneous business applications. Web services provide a standardized framework for describing, searching
Figure 2.4: A composite Web service created by a service registry.

and interacting with online applications. These applications are self-describing, lightweight, inter-connectable, and are mostly based on open XML standards, which allow Web services to interoperate and communicate across different platforms and programming languages. The World Wide Web Consortium (W3C) has defined standards for message formats and communication protocols so that Web service developers can follow. Therefore many software vendors have since introduced development environments such as Microsoft.NET framework and Java Web services toolkit to ease the developing process by encapsulating the details of message formatting and service binding.

Service composition refers to combining existing services to create larger and more complex services. One way to create composite services is through the use of interactive tools such as Agent Service Komposition Interface Tool [19] (ASKIT) or TRIANA [25]. These tools hide the low-level details of service composition to allow developers to focus only on the design of workflows using simple menu selection. However, these tools require developers to manually compose their desired services and this may introduce unnecessary dependencies between the component services. In order to overcome these dependencies, tools such as SHOP2 Hierarchical Task Network (HTN) planner [19] uses Artificial Intelligence (AI)
planning technique to generate composite services semi-automatically. SWORD [21] uses a rule-based expert system to determine whether a desired composite service using existing services can be realised, whereas Self-Serv [5] adopts a peer-to-peer (P2P) approach to compose services.

2.1.4 Description of atomic services

Web services address the need for interoperability by defining a standard XML-based protocol, called Simple Object Access Protocol [15] (SOAP), to model interactions between them. These interactions typically involve method invocations exchanged as either synchronous (HTTP) or asynchronous (SMTP) messages. These messages must adhere to SOAP’s formatting rule. Given a particular input, the application will produce a certain output. This allows all actors to know how to process the messages. SOAP provides flexibility by allowing extensions to be added to the header element and using custom data types in its payload. Furthermore, its information encoding is internationally recognized and platform-independent. Due to its simplicity, flexibility and power, SOAP is now widely accepted as the standard for Web services’ communication.

Even though SOAP successfully allows all Web services to exchange messages in a common language, it does not explain how to interact with these services. Web Services Description Language [17] (WSDL) was then introduced to describe these services. WSDL separates the abstract descriptions of a service’s functionalities from its concrete endpoint addresses mappings. The abstract descriptions include the operations and their input and output messages. A portType groups a set of operations supported by any number of providers providing a common service. An operation can be viewed as a method signature, it describes the required inputs and its corresponding outputs. The concrete aspects of WSDL define services as collection of ports. Each port represents a physical location using a specific binding to a particular service. Binding maps operations and messages of a portType to a particular communication protocol and data encoding format. Although communication protocols in WSDL are extensible, only SOAP, HTTP GET/POST, and MIME are recognized. Currently, WSDL provides support for a range of message interaction patterns. It supports one-way input messages that have no response, request and response two-way messages, and one-way messages with or without a response. The last two interaction patterns allow a service to specify other services that it needs.
2.1.5 Description of composite services

Microsoft’s XLANG and IBM’s Web Services Flow Language [20] (WSFL) are some earlier works used in designing business processes by combining Web services. XLANG focuses on creation of business processes and the interaction among Web services. XLANG includes robust exception handling and supports long-running transactions but not recursive composition. Recursive composition uses existing composite services as components for other service compositions. On the other hand, WSFL considers two types of service compositions. The first type describes the execution sequence of functionalities provided by composite services while the second type describes the communication between composite services. WSFL exposes a WSDL interface to allow recursive composition. Like XLANG, WSFL also supports exception handling, but it does not support transactions.

Business Process Execution Language For Web Services [8] (BPEL4WS) is a combination of both XLANG’s and WSFL’s features. For simplicity, we will use BPEL for BPEL4WS. BPEL enables developers to define the structure and behaviour of a set of Web services that jointly implement a shared business solution, namely a business process. Each element of the set of services defines its interface using WSDL and the business process is itself a Web service. A BPEL business process supports HTTP or SOAP messages and defines its own interface using WSDL as well. Therefore the interactions between a process instance and each service element can be either synchronous or asynchronous. In a hierarchical point of view, BPEL is layered on top of WSDL and the following sentence explains its relationship with WSDL. WSDL provides description on how and where to interact with a service whereas BPEL defines the order of sequence for these interactions to occur.

BPEL introduces three constructs to support the structure, information, and behaviour aspects of service composition. A partnerLink describes a named association between the composite service and a constituent Web service and it is being defined in both of their WSDL documents. The partnerLink concept and the WSDL descriptions define the types of services and the types of messages they exchange that combine to form the service composition. BPEL also defines the concept of a variable to describe the information aspect. The composite service defines a set of variables, with each of them capturing the current state of a specific service. This defines what messages has been sent and received.

BPEL describes the behaviour of composite service using the concept of an activity. A BPEL defined service is basically a set of activities or “steps”. BPEL considers service invocations as basic activities that are handled using the receive, reply, and invoke tags. Other basic activities such as wait, assign, throw,
terminate, and empty allow processes to wait for some time, copy data from a location to another, indicate error conditions, terminate entirely, and do nothing respectively. The basic activities can be combined to create arbitrarily complex business processes using structured activities. Structured activities such as sequence, flow, while, switch, and pick organizes the interactions between the process instance and its partnering component services into sequential, parallel, repetition, selection, and triggered selection flows respectively. Structured activities can be used concurrently to enable business processes to be reusable and scalable.

BPEL supports both hierarchical and conversational composition patterns. Variables in BPEL are used as containers for input and output messages during service invocations, and to track the status of parallel branches. BPEL describes the service that invokes the process or any service that the process invokes as partners. Partners need to be defined in both the BPEL and WSDL documents. BPEL supports partnerLink(s), myRole, and partnerRole. WSDL supports partnerLinkType, roles, and the portType associated with a role.

Web Ontology Language for Services [14] (OWL-S) is an OWL-based ontology language that is capable of specifying both atomic and composite services. It consists of well-defined semantics for describing complex logical and taxonomical relationship between Web services. The fundamental idea of semantic Web is to make applications or agents produce reliable and large-scale interoperable composite services. This requires the descriptions for Web services to be interpretable by computers. However, contents in today’s Web are in natural language for human beings to interpret, hence semantics are not provided. The introduction of OWL-S is to bridge the gap between current Web services and the semantic Web. OWL-S is motivated by the needs of providing three essential piece of information about a service; what it does, how to access it, and how it works.

Although both OWL-S and BPEL provide ways to describe composite services, there are some key differences between them. BPEL lacks the ontology-based semantics whereas OWL-S provides formal semantics. However, BPEL supports fault-handling, execution monitoring and transactions, all of which OWL-S is unable to support.

2.1.6 Publishing and discovery of Web services

Often it is useful to collect metadata about a collection of services and make this information available in a form that is searchable. Such metadata aggregation services are a useful repository in which organizations can publish the services they provide, describe the interfaces to the services, and enable domain-specific
taxonomies of services. The service registries, shown in Figure 2.2 are repositories that allow Web services providers to publish their services to potential requesters. Service requesters can then search in these registries when they require a service. Service registries provide information about the location of the required service and the procedures to access it. In other words, a service registry provides services like a yellow-pages directory. Organizations advertise their services in the directory so that potential consumers can discover and contact them. All the consumers need to know is the type of service or the name of the service provider and they will be able to find out the information required to contact the service from the directory.

Universal Description Discovery and Integration [11] (UDDI) registry is an example of the available service registries. UDDI offers a systematic way to find service providers similar to Internet search engines such as Google.com and Yahoo.com. They can also be extended to allow developers to search for existing services to collaborate in their service composition during runtime. In the service composition scenario, the developer describes the interface it requires and searches for a component service that meets the functional requirements or is provided by a well-known partner.

2.1.7 Execution of Web services

At present, Web services are mostly executed using software applications from Microsoft .Net framework or Java Web services toolkit. A client program reads in the WSDL documents that it retrieves from the UDDI registries and translates the definitions into corresponding programming languages. It then builds the appropriate proxy and stub code to communicate with the server. Applications such as Microsoft Visual FoxPro and Java Web service client generally regard the execution of composite services as a “do or fail” task, a single problem with any of the component services during execution will cause the entire service request to fail. In short, these applications do not offer any form of failure recovery to counter the common failure and disappearance of Web services in the open system environment.

2.1.8 Why JACK Teams is better in executing Web services?

In order to overcome possible failure and disappearance of services during execution, composite services should be developed as teams. We will investigate the reliability of using team-oriented structure in composite service creation and execution. JACK Teams [19] is an existing, powerful Beliefs, Desires, and Inten-
tions (BDI) agent team development environment that provides a team-oriented modeling framework. The main concept of JACK Teams is to encapsulate coordination activities by assigning roles to each team member. Although all members are coordinated by being given a specific task, each member is responsible for determining how to achieve its specific goals. JACK Teams’ goal-oriented execution engine ensures all possible plans are tried before failing a request at both the agent and team level. Therefore, JACK Teams systems are robust and persistent. In Section 3.3, we will further demonstrate the potentials of using JACK Teams to achieve robustness in service composition.

2.2 Existing error recovery techniques

There are two basic views of Web services, namely, Transactional Attitudes (TA) and Coordinated Atomic (CA) Actions. Transactional attitudes take the traditional distributed system approach, treating atomic Web services as transactions. Coordinated atomic actions on the other hand, views each Web service as autonomous actions that can be coordinated to achieve complex tasks. These two different views lead to two different approaches towards composite service error recovery, which will be discussed in the following subsections.

2.2.1 Backward error recovery based on transactional attitudes

Backward error recovery [12] is considered as the only “real” recovery function as all unexpected effects of error are totally removed by rolling the affected system back to its previous correct state. This recovery approach is widely employed in distributed databases for recovering from simultaneous transactions and is built around the ACID properties. However, despite its advantages, backward error recovery can not be applied to all types of services. Systems that involve human beings, movement of goods, operations on the real-world environment, and real-time systems do not have time to go back.

Transactions - Atomicity, Consistency, Isolation, and Durability (ACID) properties

In 1983, Haerder and Reuter [13] presented guidelines on how to prevent multiple transactions from being corrupted by hardware, database, or operating system failures in the form of Atomicity, Consistency, Isolation, and Durability (ACID)
properties. This proves to be most successful in guaranteeing the reliability of transactions for any system that conforms to their principles:

- Atomicity. If a transaction succeeds, every task within the transaction must succeed.
- Consistency. If integrity constraints on the database are specified, any transaction that violates these constraints is considered invalid.
- Isolation. Every task in a transaction can function on its own without disclosing its internal information to other tasks.
- Durability. Once a transaction is committed, the state of the database can not be changed unless another transaction is issued.

How transactions relate to Web services composition?

The ACID-style transaction has been the focus for transactional models to follow and it has been widely used for both local and distributed closed transaction systems. ACID-style transactions are also employed extensively to enforce the fault tolerance for atomic Web services, but they are deemed to be unsuitable in enforcing reliability for Web service composition [10]. Firstly, the transactional support of the Web services needs to cooperate among them in order to manage the transactions between the constituent services in a service composition. This is unlikely because the transactional support may not be compliant with each other due to the autonomy of the Web services. Furthermore, the administrative domain of these services is not compelled to develop standardized transactional support. Secondly, transactions work on the basis of locking participating resources until the embedding transaction is terminated or committed. This is unacceptable for Web services as locking their resources indefinitely could meant that their concurrent clients could be experiencing long delays before their requests can be handled.

An improved transactional model, namely the split model, overcomes the issue of increased access latency for composite Web services by separating the embedding transactions into a number of concurrent sub-transactions that can commit independently. This prevents the need to lock resources of the constituent services in a service composition before the embedding transaction commits or terminates. These sub-transactions are simply transactions that are already supported by each individual Web service. Hence the amount of time required to commit the overall transaction of a composite service will only take at most the longest time required to commit the sub-transactions.
In order to satisfy the atomicity property of transactions, either all or none of the sub-transactions in the overall transaction can be successfully committed or terminated. Therefore, in the event of a sub-transaction failure when others had already committed, compensation operations should be used to roll back all committed sub-transactions to their previous correct state.

However, in order to support this compensation recovery scheme, all participating Web services must provide compensation operation for each functional operation they offer. The intrinsic autonomy of the Web services makes it difficult to expect every service to implement compensation operations for all their functional operations. Moreover, there are no common transaction semantic, context representation, and coordination means for transactions to participate in a Web services environment. Since service consumers can neither discover whether compensation operations are provided at the other end nor can they find out the transactional semantics of the services they are invoking, a lot of ad hoc programming work will be required to compose Web services. One way to address these concerns is by adopting the concept of *transactional attitudes* for all Web services.

**Transactional Attitudes**

The concept of transactional attitudes allows Web service providers to declare their individual service’s transactional capabilities and semantics, and also allows Web service consumers to declare certain transactional requirements for their desired services [18]. Provider Transactional Attitudes (PTAs) and Client Transactional Attitudes (CTAs) are mechanisms that facilitate these explicit transactional semantics descriptions and they are provided in the Web Services Transactions (WSTx) framework.

By allowing transactional semantics to be described explicitly, PTAs can be used to automate the composition of individual transactional services into larger transactional patterns, while maintaining the autonomy of each service. This saves clients some effort during service composition as the clients can avoid ad hoc programming of constituent services that may have different transactional semantics, which is impossible to find out if it can not be described explicitly. A PTA includes the name of an abstract transactional pattern together with any port-specific information that is required to make the pattern complete. The abstract pattern represents a well-defined state-machine describing valid transactional states, state transitions, transition-triggering events. Each abstract pattern refers to certain transactional behaviour that is common to many Web services.
CTAs allow the clients to state which constituent Web services are vital and which are not critical to the success of the overall composed transaction. Without the CTAs, clients would have to manually program the use of Web services to ensure that the desired outcome is reached and this can be a very complex process. A CTA is established by using a certain WSDL port type to manage Web transactions. This port type implies some pre-defined transactional pattern. Within the overall composed transaction, the client would execute one or more named actions, where each action represents a provider’s transactional service. The client initiates an action by binding to an action port. Each action port serves as a proxy to a participating provider’s transactional port and represents a unique provider transaction executing within the overall composed transaction. Hence by binding to an action port, the client would be able to invoke the provider’s service with the correct client transaction. When using the action port, the client may only invoke forward operations and not completion operations such as commit, abort, and compensation operations. The flexible atom attitude describes a set of client actions being grouped into an atomic group that can have one of a set of defined group outcomes. The client specifies through the FA what the acceptable outcomes are as an outcome condition, which is described in terms of the success or failure of the individual actions. After executing the forward operations of these actions, the client can then request the completion of the FA according to the outcome condition through a CTA port. Each CTA port manages Web transactions associated with a specific CTA and is used to invoke the completion operations involved in the transactions.

Web Services Transactions framework

Currently, the WSTx framework defines three patterns of PTAs:

1. Pending-Commit,
2. Group-Pending-Commit, and
3. Commit-Compensate.

These PTAs describe transactional behaviour that is common to many Web services. It is expected that as more transactional behaviours emerge, the number of PTAs supported by WSTx framework will also be increased.

The Pending-Commit attitude describes the transaction of a single Web service when the effect of invoking a single forward operation remains in a pending state until an acceptance or a rejection of the operation. Forward operations
are represented in the WSTx framework with a `<wstx:forwardOperation>` element. The effect of a forward operation is brought to a pending state if it is invoked successfully, else the effect is rejected. The effect of a forward operation can be explicitly accepted or rejected by invoking a commit operation or an abort operation respectively. The commit operation is represented in WSTx framework with a `<wstx:commitOperation>` element and the abort operation is represented using a `<wstx:abortOperation>`. WSTx framework provides the `<wstx:simpleCorrelation>` input/output extension element to link the forward operations with the commit or abort operations. This element allows the service provider to specify the message parts for the correlation encoding.

The Group-Pending-Commit attitude describes the transaction of a single Web service where the effect of invoking a group of forward operations can be held in a pending state until the invocation of a commit or an abort operation. A new group of operations can be created explicitly with `<wstx:beginOperation>` element or implicitly with `<wstx:forwardOperation>`. The group of operations is brought to a pending state by invoking a single prepare operation, which is represented by `<wstx:prepareOperation>` in the WSTx framework. However, the group of operations is brought to a pending state only if the prepare operation is invoked successfully. Otherwise, it will be rejected. The group of forward operations is accepted by invoking the `<wstx:commitOperation>` and rejected by invoking the `<wstx:abortOperation>`. A Group-Pending-Commit attitude must include a correlation encoding to link individual operation with the group of operations in a single service as well as the abort and commit operations. This correlation encoding is provided by using the `<wstx:simpleCorrelation>`, which is similar to the Pending-Commit attitude.

The Commit-Compensate attitude describes the transaction of a single Web service for which the effect of a single forward operation is immediately accepted, but can be later reversed by invoking an associated compensation operation if required. The compensation operation is represented in the WSTx framework as an `<wstx:compensationOperation>` element. Once again, a Commit-Compensate attitude must include a correlation encoding to associate a forward operation with a compensation operation and this correlation encoding is similarly provided by the `<wstx:simpleCorrelation>` element.

The client acceptable outcome is specified in the WSTx framework as an outcome condition in the flexible atom attitude. In order to ensure the reliability of the execution of the overall transaction, a WSTx middleware is required as well. Once all forward operations are invoked, the client may request the completion of the flexible atom according to the condition. The WSTx middleware will then attempt to satisfy the outcome condition by invoking the appropriate comple-
tion operations on the service providers. This requires the following monitoring components to exist in the WSTx middleware:

1. a persistent log of all ongoing transaction and their state,
2. a persistent log of all relevant transactional-state transformations for each global transaction, and
3. a time service to detect timeouts in transactional interactions with participants.

Although the design of the WSTx framework uses standard Web services technology in WSDL and the WSDL extension mechanisms only, it is also open to accommodate emerging technologies in the Web services community such as Business Transaction Protocol (BTP), Tentative Hold Protocol (THP), J2EE Activity Service specification, and XML Transactioning API for Java [18]. This is mainly due to its ability to isolate the transactional properties of Web services from other aspects of their service description. Furthermore, it does not require an agreement by transactional participants in a service composition on a common semantic, representation, and coordination protocol. The implementation of the WSTx middleware can also be easily realised by using existing Web Application servers such as IBM’s Web Services Toolkit [18].

2.2.2 Forward error recovery based on Coordinated Atomic Actions

Forward error recovery, which uses exception handling mechanisms, is commonly employed in the specification of Web service composition to handle the event of errors [10]. Usually the state of the system from which error is being recovered is non-optimal. This is due to the unexpected tasks that are activated to recover from these errors. However, forward error recovery is the only approach used to recover from failure in critical systems such as nuclear power plants, aircraft systems, and chemical plants.

Exceptions handling

Exception handling is the method of building a system to detect and recover from exceptional conditions. Exceptional conditions are any unexpected occurrences that are not accounted for in a system’s normal operation. It is difficult to protect a system from the effects of exceptional conditions because we can never
be able to anticipate all unexpected occurrences when designing the system. If these exceptional conditions were not properly caught and handled, they can lead to application failures and system crashes. Unfortunately, no well-defined techniques exist for building robust exception handling into a system. Most methods are ad hoc and limited to what the developer team can anticipate the system will encounter. However, most of these unexpected conditions are common problems that may exist in more than one system and many of the resulting failures can be traced back to simple conditions such as checking for null pointer before referencing it, and checking for illegal argument in an input. Therefore as long as good software engineering practices such as code reviews, code walkthroughs, and thorough testing are applied appropriately, we can eliminate many of these exceptional conditions.

As aforementioned, it is unrealistic to build a system that accounts for all unexpected occurrences. Therefore it is necessary to build default exception handlers that will attempt to recover from any of these unanticipated conditions. If it is a critical system or a system that has real-time deadlines, some form of graceful degradation must be executed in the event of unanticipated conditions to reduce the damage done by any system failures. This form of graceful degradation can be provided by either programmed exception handling or default exception handling. Programmed exception handlers are mechanisms built into software for specific exceptional cases that are known to occur. Since these cases are better understood by the system developers, protection for them can be incorporated easily into the systems. As for the rest of the exceptional conditions that are not anticipated, default exception handlers must be built. These default handlers may be built within the programming language or operating environment itself, transparent to the system developers. In order to achieve robust operation, as much exception handling as possible is desired. However, exception handling overhead may be too great for real-time systems and make timing and scheduling difficult.

How exception handling relate to Web service composition?

One specification of composite Web services that exploits exception handling mechanisms is BPEL. The fault handlers of BPEL can be associated to an activity such that the respective handler is activated and the execution of the activity is terminated when an error occurs. However, when an activity is defined as a concurrent process and at least one constituent activity signals an exception, all other constituent activities will be terminated as soon as one exception is caught, but only the handler that caught the exception will be executed. This poses a problem as the rest of the constituent activities may not be able to recover.
to a correct state when other activity signals an exception. This disadvantage of BPEL applies to all XML-based languages for Web service composition that supports concurrent activities and exception handling. One solution to overcome the abovementioned disadvantage is to apply the concept of coordinated atomic actions that provide a sound method of handling simultaneous exceptions.

Coordinated Atomic Actions

The coordinated atomic action concept is introduced as a unified general approach to structuring complex parallel activities and supporting error recovery of multiple interacting participants [10]. It provides a conceptual framework for dealing with cooperative and competitive concurrency and for achieving fault tolerance by extending and integrating two complementary concepts in atomic actions and ACID transactions. Coordinated atomic actions have the characteristics of both concepts: Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whereas ACID transactions are used to maintain consistency of shared resources in the presence of failures and competitive concurrency. This allows failure recovery for different components participating in the coordinated atomic action execution.

Each coordinated atomic action is designed as a set of atomic actions participating and cooperating in the coordinated action. The action starts when all roles have been activated and completes either when no error has been detected, or after a successful recovery, or when a failure exception has been propagated to the containing action. More specifically, when several errors are detected inside an action concurrently, all the participants will be involved in the coordinated recovery. If these errors are recovered successfully, the action completes normally. Otherwise, the errors will be propagated and the responsibility of recovering from them is passed on to the caller of the action, while all transactions on external resources are aborted. External resources that maintain ACID properties can be accessed simultaneously by several coordinated atomic actions. A coordinated atomic action execution looks like an atomic transaction to the outside world. The state of the coordinated atomic action is represented by a set of local and external resources; the coordinated atomic action deals with these resources to guarantee their state restoration.

The coordinated atomic action concept allows designers to reduce system complexity by nesting several actions into one with well-defined rules [22]. Participants of a nested action must participate in the containing one, a participant can only be in one sibling action at any time, the containing action can complete only after all of its nested actions have completed. When an action fails to re-
cover from an error or fails to deliver the required results, the failure exception is propagated to the containing action. Hence transferring the responsibility for recovery to a higher level and leaving participants involved in the action execution to facilitate subsequent recovery at a higher level.

**Web Service Composition Action framework**

Although coordinated atomic actions do provide a sound basis for dealing with simultaneous exceptions, the need to maintain ACID properties over external resources, which in this case corresponds to Web services, are not suited for dynamic open environments. Hence Web Service Composition Action (WSCA) framework is then introduced to adapt the coordinated atomic actions to the specifics of Web services. The main difference between WSCA and Coordinated Atomic Actions is that WSCA relax the transactional requirements over the external resources.

WSCA participants define interactions with the constituent Web services by describing the role of each service in the service composition. Like coordinated atomic actions, WSCA views each Web service as an external resource. However, WSCA do not require all interactions to be transactional. Instead the interactions may be transactional only if the operations of the invoked component services are. Every WSCA participant must also describe actions to be taken when the component services signal an exception. This exception may be either handled locally by the participant or be propagated to the containing WSCA. The latter leads to cooperative exception handling specified in the containing WSCA. When Web services involved in the cooperative exception handling offer compensation operations, the WSCA participant may exhibit a relaxed form of atomicity. If an external resource is compensated successfully, the action itself is considered to be atomic as the external resource restores its initial state. However, any other actions that accessed a Web service before it is being compensated may have invalid values.
In this chapter, we have gathered key properties from both the transactional attitudes view and coordinated atomic actions view of handling failures in service composition. However, backward or forward error recovery alone is not suitable in supporting robust creation and execution of composite services. An investigation in the agent research for building robustness motivates us to look into BDI team-oriented architecture for reliable composite services creation and execution. JACK Teams as a BDI team-oriented language framework is considered and discussed with an emphasis on its ability to perform error handling. The properties gathered from WSCA and WSTx standards are used as guidelines to evaluate the fault-tolerance support in JACK Teams. We then proceed to describe how JACK Teams recovers from failure using a sample JACK Teams program. Lastly, we propose a new economical failure recovery by extending and adapting the default behaviour in JACK Teams to accommodate the needs of composing Web services.

3.1 Why backward and forward error recovery alone is unsuitable?

Table 3.1 summarised essential properties gathered from both WSTx and WSCA frameworks that support composite services failure recovery. Although backward error recovery, specified in terms of WSTx framework and built upon transactional attitudes, satisfies most of these requirements, its strict compliance of the ACID properties requires all services involved must be transactional. A transactional service is one which executes either completely or not at all. The results of a transactional service are usually stored in a database management system. Each transaction is treated as a unit of interaction with a database independent of other transactions and the system’s states must be logged for every transactions so that any reversal of a transaction can be performed by rolling back to the system’s previous correct states. A non-transactional service, on the other
hand, has no guarantees about successful completion nor about the ease of undoing the effects after unsuccessful or incomplete execution. It may sometimes results in a change to its environment. An example of a non-transactional service is an Internet-controlled device such as the robotic telescope in the University of Bradford Website, while examples of transactional services are online shopping, flight booking, and hotel accommodation reservation services. As a result, backward error recovery can only support failure recovery for composite services whose constituent services are transactional and cannot guarantee the reliability of non-transactional composite services.

<table>
<thead>
<tr>
<th>Sno</th>
<th>Suitable properties</th>
<th>Provided by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exception handling</td>
<td>WSCA</td>
</tr>
<tr>
<td>2</td>
<td>Description of least acceptable outcome conditions</td>
<td>WSTx</td>
</tr>
<tr>
<td>3</td>
<td>Rolling back to previous correct state during failure</td>
<td>WSTx and WSCA</td>
</tr>
<tr>
<td>4</td>
<td>Maintaining ACID properties in simultaneous task execution</td>
<td>WSTx and WSCA</td>
</tr>
<tr>
<td>5</td>
<td>Coordinated execution and failure recovery</td>
<td>WSTx and WSCA</td>
</tr>
</tbody>
</table>

Table 3.1: Properties required for composite service failure recovery

Albeit forward error recovery alone is only capable of rolling a system back to a sub-optimal state instead of its previous correct state, coordinated atomic actions exploit forward error recovery at the service composition level and also incorporate backward error recovery at individual services. As mentioned in Section 2.2, the WSCA framework relaxes the atomicity property in its transactions so that component services need not provide compensation operation for every functional operations. Hence the WSCA framework is considered to be a better solution as it offers failure recovery support for both transactional and non-transactional services. This shows that in order to provide failure recovery for composite services, it is necessary to satisfy the properties of both backward and forward error recovery.

Both frameworks remain at the stage of initiatives and waiting for adoption as standards. Standards become technologies when implemented. At the time of writing, there is no single technology that supports the implementation of both WSCA and WSTx. On the contrary, agent technologies, especially BDI-based teamworks have been proven to be particularly suitable for distributed robust multi-agent system implementation. They may not strictly conform to the above error recovery standards, but they do offer properties that satisfy most of the requirements listed in Table 3.1. Especially, both WSTx and WSCA require substantial modification in the existing Web service standards such as WSDL. The development of reliable composite services using existing standards become almost impossible. We will show how BDI-based teamwork work on existing
Web services and at the same time ensure robustness and reliability in service composition.

3.2 Hierarchical team architecture technologies in agent community

JACK Teams [4] and Teamcore [24] are examples of technologies in the agent community mentioned in Section 3.1. Teamcore, revised based on STEAM, builds up a complex hierarchical structure of joint intentions, individual intentions, and beliefs about others’ intentions using a bottom-up approach. The team is aggregated from the joint intentions and joint beliefs shared by the team members. There is no explicit implementation of a team entity. The coordination is achieved through reasoning about joint intentions and joint beliefs by the team-members. This explains the bottom-up team formation. On the contrary, JACK Teams uses a top-down approach in its team formation. Instead of sharing goals amongst its members, a centralised team entity explicitly programmed to control the team goal and execute the teamplans. The team entity then coordinates the team members in performing their tasks to achieve the team goal [19]. As these models are hierarchical, the virtual entities in Teamcore and the team members in JACK Teams may themselves be teams. This is illustrated in Figure 3.1.

Padgham and Liu [19] suggested that Teamcore would not be suitable for the type of collaboration required in composing Web services. Web services are autonomous independent software system implemented by various providers, it is then unrealistic to impose joint intentions and expect them to intentionally
carry out collaborations. Instead, they are more suitable to be treated as task
performers, which may require explicit managerial entities to allocate and control
the execution of sub-tasks. Therefore, the top-down approach implemented in
JACK Teams is considered to be more suitable for service composition. In the
following section, we will look at JACK Teams language constructs with a special
focus on exploring how error handling can be achieved using JACK Teams.

3.3 Concepts of JACK Teams and how it meets the needs
of composite service failure recovery?

JACK Teams is an extension of JACK Intelligent Agents Language [3], which pro-
vides agent-oriented programming constructs for developing agent applications.
The JACK Agent language is built on top of Java. JACK Teams further extends
the JACK Agent language by providing team-oriented programming constructs
and support. Some main JACK Teams concepts are explained in the following
sub-sections. This is followed by a discussion on how JACK Teams satisfies the
requirements to support failure recovery for composite services.

3.3.1 Team Entity

The team entity is the core concept in JACK Teams. Like a JACK agent, a team
is also an individual reasoning entity with its own beliefs, desires, and intentions.
In addition to normal knowledge-building, a team entity is also concerned with
the coordination of its sub-teams. This is done by choosing the teamplan to
handle a particular event. JACK Teams introduces the teamplan construct to
specify team-oriented behaviour. As JACK Teams extends the JACK Agent
language, a team can use agent plans as well. The team entity is also responsible
for persisting in realizing the overall team goals even when the selected plan fails.

3.3.2 Team

A team is defined by the roles it may perform for others and the roles it may
require other sub-teams to provide. The formation of a given team is achieved by
attaching sub-teams that are able to perform the roles required by the team. A
team does not possess any actual sub-teams, instead it is automatically provided
with role containers to hold actual role/sub-team selections. A team construct
contains a set of teamplans or plans that perform tasks to achieve specific goals.
These sub-teams can then be added or removed dynamically during the team establishment phase when a teamplan is executed.

3.3.3 Sub-team

A sub-team can be either an actual agent with no sub-teams or a team that comprises of other sub-teams. Hence a team's structure is not restricted to a two-level design but rather it can be a multi-level hierarchy. Although sub-teams act in accordance with the goals set by the team entity, they are individually responsible for accomplishing these goals.

3.3.4 Role

A role in JACK Teams establishes the link between a team/sub-team relationship. A team entity requires a role. A sub-team who declares that it can perform the role becomes eligible for selection. This description is expressed in terms of the goals the role is responsible to achieve or the events the role can respond to.

3.3.5 Teamplan

A teamplan is essentially a set of steps describing how a particular task can be achieved in terms of one or more roles. It contains methods to establish the sub-teams required to perform the roles for a certain task and to coordinate the sub-teams in achieving their own specific goals. The coordination of sub-teams to achieve specific goals is supported by the \texttt{@team\_achieve} and \texttt{@parallel} statements. These statements are specified in the body reasoning method of a teamplan. The \texttt{@team\_achieve} statements activate a sub-team by triggering an event in the sub-team. It is simply a sequential coordination of a sub-team's behaviour, the team that triggered the \texttt{@team\_achieve} statement will wait till the event had been processed by the sub-team. On the other hand, the \texttt{@parallel} statement allows simultaneous sub-tasking of a set of statements within itself. Similarly, the team that is running the \texttt{@parallel} statement will be suspended while all enclosed statements are executed in parallel.

3.3.6 Agent-oriented concepts

Events and plans are two important concepts inherited from the JACK Agents language [3]. Events are programming constructs that motivate a team to take
action. They can be regarded as the origin of all activity within JACK. Hence a team remains idle in the absence of events. Whenever an event occurs, a team initiates a task that causes the team to select one of its available plans, executes it until it succeeds or fails. An event is said to have succeeded if the plan execution succeeds, otherwise it is considered to have failed. Normal events do not persist after the first instance of plan failure. On the contrary, BDI events enable teams to choose a number of plans for execution and these plans are attempted in turns. Only when the entire set of plans is exhausted, then the event is considered to have failed. A method signature and its method body in a Java program presents a simple analogy to an event and its plans in JACK Teams.

3.3.7 How JACK Teams satisfies the required properties for composite service failure recovery?

As aforementioned, Table 3.1 provided a list of necessary requirements to meet in order to handle failures in composite services. Even though JACK Teams may not conform to either the WSTx or the WSCA standards, its properties encourage its application in composite service creation and execution. The following points describe how JACK Teams satisfies the requirements in Table 3.1:

1. A team in JACK Teams may throw a Java exception when handling an event allocated either with a @team_achieve or @parallel statement. The handling of exceptions follows strictly to the Java model for exceptions [4] and it can be provided in JACK Teams through the reasoning method fail() in either plans or teamplans. If an exception thrown is not caught within the team, it will be propagated back to the top-level team who executes the @team_achieve or @parallel statement.

2. Although JACK Teams is unable to explicitly describe the least desirable outcome conditions like the WSTx framework, the ordering of available teamplans offers another way to achieve similar effects. That is to declare teamplan with the most optimal outcome conditions first, followed by, other teamplans with less desirable outcomes. As by default, the team selects the teamplan according to the order in which they are declared in, hence the teamplan with the least acceptable outcome condition can be declared last. This allows JACK Teams to always try to achieve better outcomes.

3. When a teamplan fails in JACK Teams, all executed steps in the teamplan are discarded. This is similar to the third requirement listed in Table 3.1 where the state of the team entity is rolled back to a state before the teamplan was executed.
4. As mentioned in Section 3.3.5, @parallel statements allow execution of parallel tasks. The task synchronisation of these statements is implicitly handled by JACK Teams language. This enables the ACID properties to be maintained when executing simultaneous tasks.

5. The team entity in JACK Teams fully satisfies the fifth requirement in Table 3.1. It coordinates the execution of sub-teams through the teamplans and it is responsible to invoke alternative teamplan if the selected one somehow fails.

Point (1) and (3) provided evidence that the failure recovery mechanism in JACK Teams is similar to backward and forward error recovery respectively. As mentioned in Section 2.2, we know that backward error recovery removes failures by rolling the affected system back to its previous correct state. The teamplans and plans in JACK Teams offer exception handling mechanism, which is the basis of forward error recovery. As we can see, JACK Teams should be able to provide both backward and forward error recovery, which are dominant recovery techniques employed in composite service recovery. In this research, we will implement JACK Teams for composing example Web services to investigate if various failure recovery techniques can be realised.

### 3.4 Providing failure recovery for service composition in JACK Teams

By default, the teamwork in JACK Teams offers persistence in achieving its goals at both individual and team level. JACK Teams can take advantages of redundancy at the team level in terms of replacing sub-teams dynamically with alternatives in the presence of failure. First and foremost, persistency at individual level addresses the issue of an operation in an individual Web service failing to perform a required task. This particular service must try all its possible plans to accomplish its specific goal before it can be considered as failed. As a result, all teams representing component services in JACK Teams must be goal-oriented.

Secondly, persistency at team level addresses the issue of a composite Web service failing to perform a user’s request. Similarly, the composite service must try all possible teamplans before it can be considered as failed. It was mentioned earlier that a component service can be a composite service as well and that there may be more than one teamplan to achieve a specific objective. Hence all teams representing composite services in JACK Teams must be goal-oriented as well.
Lastly, redundancy of team members addresses the issue of a Web service failing to perform the required task after trying all possible plans or teamplans. Any failed service should be replaced by another that can perform similar roles. This requires the team entity to explicitly access the relevant role containers to swap the team representing the failed service with another.

3.4.1 Conceptual mappings between JACK Teams and service composition

A *team* in JACK Teams is conceptually similar to a composite service, where each team member is a component service. The component service can either be an atomic service or a composite service. A composite service coordinates the interactions between the component services to accomplish some goals. This is also similar to a team in JACK Teams, who is responsible for establishing and coordinating its *sub-teams* to fill the required roles to accomplish a task.

In JACK Teams, an *event* triggers a team to take certain actions. These actions include initiating a *plan* or *teamplan* to handle that particular event. This is conceptually similar to invoking an operation in a Web service. The events in JACK Teams can be viewed as operations offered by a Web service, whereas the plans can be regarded as procedures on how these operations are realised. In addition, the port type clause in a Web service’s WSDL document aggregates a set of relevant operations. Thus a port type can be mapped as a role in JACK Teams.

A role type in JACK Teams aggregates a set of events the role can perform. Therefore, a *role container* groups teams that can perform similar roles together. This is done either *implicitly* when a *team* instance capable of performing a certain *role* is created or *explicitly* through an initialisation file. The role container concept allows swapping of sub-teams dynamically as teams that can perform a particular role can be accessed easily through it.

A *teamplan* is conceptually similar to the composition plan of a composite service in that it describes how execution is being performed and coordinated between the composite services and its constituent services. Therefore representing composite services using JACK Teams allows for more than one composition plan to be specified.
3.4.2 Persistency at individual and team level

As aforementioned, the persistency of JACK Teams allows Web services to try all possible ways to achieve their specific goals. These services should recover by invoking alternative plan or teamplan to perform the failed task. In order to support this persistency in JACK Teams, teams and their sub-teams must be equipped with additional teamplans and plans to handle each event. Furthermore, events must extend the `BDIMessageEvent` class and set their `recover behavior` to `repost` to inherit its persistent behaviour. The following code listed various JACK programming constructs required in a JACK Teams program to implement the persistent behaviour at the team level:

```java
public team Spacecraft extends Team {
    #requires role Spokesperson sp(3,3);
    #requires role Pilot pi(3,3);
    #requires role Crew cr(3,3);
    #uses plan Visit1;
    #uses plan Visit2;
    #uses plan Visit3;
    #handles event PerformVisit;
    public Spacecraft(String name){
        super(name);
    }
    #posts event PerformVisit pfv;
    public void visit(String planet){
        postWhenFormed(pfv.visitPlanet(planet));
    }
}

teamplan Visit1 extends TeamPlan {
    #handles event PerformVisit pfv;
    #posts event TeamFormationEvent tfe;
    #uses role Spokesperson sp as speaker;
    #uses role Pilot pi as pilot;
    #uses role Crew cr as crew;
    #uses interface Team team;
    #reasoning method
    establish() {
        Vector busy = new Vector();
        crew = (Crew) pickRole(busy,cr);
        crew != null;
        pilot = (Pilot) pickRole(busy,pi);
        pilot != null;
        speaker = (Spokesperson) pickRole(busy,sp);
        speaker != null;
    }
    Role pickRole(Vector busy, RoleContainer rc) {
        for (Enumeration e = rc.tags(); e.hasMoreElements(); ) {
            Role r = rc.find((String)e.nextElement());
            if(r.state == Role.ACTIVE & & !busy.contains(r.actor)) {
                busy.add(r.actor);
                return r;
            }
        }
    }
    return null;
}
```
event PerformVisit extends BDIMessageEvent {
    #set behavior Recover repost;
    String planet;
    #posted as
    visitPlanet(String p){planet = p;}
}

This program is about a spacecraft carrying a team of Martians to visit Earth. This spacecraft contains three sub-teams. Each of them is capable of performing the role of a pilot, a spokesperson, and a basic crew member. The containing team, Spacecraft, embody more than one teamplan to handle the PerformVisit event and these teamplans are selected in the order that they are declared in team entity’s definition file.

The control flow of this program begins with the team entity, which in this case is the Spacecraft team, posting the PerformVisit event to itself to set up a team goal. This will trigger itself to select its first teamplan, which in this case is the Visiti teamplan. Within Visiti, a reasoning method establish{} establishes a task team, which requires three sub-teams to fill the three required roles, namely Spokesperson, Crew, and Pilot. The three uses role statements define the sub-teams required by the Spacecraft team while the two-letter initials in each of these statements are references to the created role instances. After the task team is formed, Visiti then coordinates the behaviour of its sub-teams by posting event to them through the @team_achieve statements in the body{} reasoning method. Notice that there are more than one teamplan being declared in the Spacecraft team’s definition file, this satisfies half of the requirements in achieving persistency at a team level. The other half is achieved by extending the goal-oriented behaviour of BDIMessageEvent programming construct with the PerformVisit event. The #set behavior recover statement in the PerformVisit event defines that the default recovery behaviour for the event is to repost itself when a failure occurs.

The data flow of this program is facilitated by the instance variable of the PerformVisit event. The PerformVisit instance is responsible for passing data between the Spacecraft team and the Visiti teamplan. The entry point of this program begins when the team entity is instantiated and ends after the selected teamplan is successfully executed.
Goal persistency is further exercised at an individual team member level through the @team_achieve statements in the body{} reasoning method of the teamplan. The events that were posted to the sub-teams will trigger the respective sub-teams to select a plan to handle them. Similarly, the events handled by each of these plans are BDIMessageEvent and more than one plan must be declared for each sub-team to attain its goal-oriented behaviour.

3.4.3 Making use of redundancy

Redundancy of teams addresses the issue of a failed component service in a composite service. The composite service should recover by replacing the failed service with another that provides similar functionality. In order to provide team redundancy in JACK Teams, alternative teams must be created to provide replacements in case of failed sub-teams. The sub-team instances can be described in JACK Teams explicitly through an initialisation file:

```xml
<Team :name ''StarTrek Enterprise''
 :roles {
  <Role :type ''Spokesperson'' :name ''sp''
    fillers ( <Team :name ''Archer@%portal'' >
      <Team :name ''Reed@%portal'' >
      <Team :name ''Tucker@%portal'' >
      <Team :name ''TPol@%portal'' > ) >
  <Role :type ''Pilot'' :name ''pi''
    fillers ( <Team :name ''Archer@%portal'' >
      <Team :name ''Reed@%portal'' >
      <Team :name ''Tucker@%portal'' >
      <Team :name ''TPol@%portal'' > ) >
  <Role :type ''Crew'' :name ''cr''
    fillers ( <Team :name ''Archer@%portal'' >
      <Team :name ''Reed@%portal'' >
      <Team :name ''Tucker@%portal'' >
      <Team :name ''TPol@%portal'' > ) >
}
>
```

The team entity instance for this specific implementation is named “StarTrek Enterprise” and it requires team members who can perform the three roles defined in the initialisation file. Notice that four team instances are declared in the initialisation file for each role required. Hence if “StarTrek Enterprise” only requires two members for its initial team formation, the other two team instances can be used as replacements.

The default behaviour of JACK Teams is to choose an alternative agent plan when the selected agent plan fails, and to pick out an alternative teamplan whenever the selected teamplan fails. Even if explicit instructions enable the swapping
of teams when a team member fails, JACK Teams’s default behaviour required the entire teamplan to be re-executed all over again with the substitute agent and this includes successfully-executed steps that do not involve the failed service. As a result, time and resources will be wasted whenever a failure occurs regardless of the number of successful steps.

3.5 A new economical failure recovery proposal

It is clear that we want to maximise the usage of successfully executed steps that do not involve the failed service. However, this is not a trivial case of simply tracking the execution status of each step, it is also essential to track possible dependencies with earlier steps for each step and its association to the team member. Our proposal is to add a data structure to track dependencies between each step in every teamplan, and using this set of data structures as transaction logs so that JACK Teams can roll back to a state where previous successfully executed steps are not wasted in the process. The dependency between different steps can be defined as a causal link relationship. A causal link relation

\[ S_i \xrightarrow{P} S_j \]

serves as a means for pre-condition establishment between steps, where condition P must always hold true for step Sj to take place [27].

For example, imagine a situation where a user wishes to reserve a flight ticket, book a room, and rent a vehicle. Suppose that each of these individual Web services are represented by a team. The only condition for this scenario is that the proximity of the car depot and the reserved hotel must be close enough before the user commit the room reservation. This will then require a street directory Web service that returns the distance between any two locations. Figure 3.2 illustrates the interactions between the containing team and its sub-teams in a teamplan. Suppose that the first five steps were executed successfully, but step (6) fails as all the vehicles in the car rental team are fully booked. Although the ideal solution would be to replace the failed car team with another, but swapping of teams meant that some of these steps must be re-executed. Clearly, referring to Figure 3.2, step (4) and (6) must be re-executed as they are operations performed by the failed car team.

Next, we look at the dependencies for each step in this scenario. We can see that step (5) is directly dependent on step (3) and (4) as the result of both steps make up its parameters. From Figure 3.2, we can also see that step (3) depends on the result of step (2), and step (6) depends on the result of step (5). Consequently, we can infer the following relationships:
1. step (1) is independent of other steps,
2. step (2) is independent of other steps,
3. step (3) is dependent on step (2),
4. step (4) is independent of other steps,
5. step (5) is dependent on step (4) and (3) directly, and is indirectly dependent on step (2) as well,
6. step (6) is dependent on step (5) directly, and is indirectly dependent on step (2), (3), and (4) as well,
7. step (7) is dependent on step (1), (2), and (6), and is indirectly dependent on the other three steps.

When a failure occurs in step (6), we shall first mark step (4) as \textsc{NOT\_EXECUTED}. Next, we mark step (2), (3), and (5) as \textsc{NOT\_EXECUTED} as well because step (6) depends on their output to execute. Hence when a duplicate teamplan is selected, only step (1) need not be re-executed. As for step (7), it is not executed since the previous teamplan did not proceed as far as step (6). This set of recovery steps is similar to an algorithm described by Padgham and Liu [19] and it is listed as follows:
1. Identify the last step to be executed in the team plan.

2. Change the status of all steps which directly involve the failed team to \texttt{NOT EXECUTED}.

3. Change the status of all steps which depended on the output of steps involving the failed team to \texttt{NOT EXECUTED}.

4. Swap the failed team with another team capable of performing similar roles.

5. Re-execute all steps in the team-plan that are labeled as \texttt{NOT EXECUTED}.

However, we notice that this scenario only invokes the individual services sequentially. For simultaneous task execution, we will need to incorporate the same set of recovery steps for all \texttt{@team_achieve} statements within the \texttt{@parallel} constructs.
CHAPTER 4

Implementation

Imagine a travel agency, who would like to build a composite service based on atomic services provided by other service providers. This may include a flight reservation service, a hotel accommodation reservation service, a car rental booking service, and a street directory service. The following software are used for the implementation described in this chapter.

In this chapter, we will demonstrate how JACK Teams can be used to implement the composite service and how various levels of failure recovery can be achieved in this travel agency scenario. For testing purposes, the atomic services are developed using Apache Axis 1.1 and deployed on Apache Tomcat Server 5.5.

4.1 Creating and deploying component services

In order to simulate atomic service failure, we created the atomic services required for the travel agency composite service ourselves. First, Java interfaces are created to describe the operations each atomic service will provide. Next, we generate WSDL documents from the compiled interfaces using the Java2WSDL tool in Axis. An example invocation to produce WSDL files from the compiled Java interfaces is as follows:

```
```

Where:

- `-o` indicates the name of the output WSDL file.
- `-l` indicates the location of the service.
- `-n` is the target namespace of the WSDL file.
• -p indicates a mapping from the package to a namespace. There may be multiple mappings.

In order to deploy and invoke the component services, the server skeleton classes must be built using the following command:

```java
java org.apache.axis.wsdl.WSDL2Java --server-side --skeletonDeploy true flight.wsdl
```

In addition to generating the server skeleton classes, a `deploy.wsdd` file and a `undeploy.wsdd` file are created as well. These implementation template files provide guidelines for the service writer to implement methods for the actual services, whereas the `.wsdd` files are used to deploy or undeploy the services once their methods are implemented, compiled, and made available to the Axis engine. Table 4.1 describes what server classes are generated by the Axis `WSDL2Java` tool from a WSDL document:

<table>
<thead>
<tr>
<th>SNO</th>
<th>WSDL clauses</th>
<th>Java class(es) generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For each Binding</td>
<td>A skeleton class, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An implementation template class.</td>
</tr>
<tr>
<td>2</td>
<td>For each Service</td>
<td>A deploy.wsdd file, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A undeploy.wsdd file.</td>
</tr>
</tbody>
</table>

Table 4.1: WSDL2Java server-side bindings conversions [2]

### 4.2 Building client stubs from WSDL

In order for JACK Teams to invoke Web services, proxies for these services must be developed. These proxies can be generated in the form of Java classes by a WSDL2Java tool using the services’ WSDL files. Table 4.2 shows the Java files that may be generated from a WSDL document. However, generating proxy classes for Web services alone does not allow invocation of services in JACK Teams. The generated classes must be incorporated into the plans of a team that represents a specific Web service. This is described further in Section 3.3.

After obtaining the WSDL file for each required component service, a set of Java classes are generated from these WSDL files using the Axis `WSDL2Java` tool. The following command produces these client-side bindings:

```java
java org.apache.axis.wsdl.WSDL2Java flight.wsdl
```
Table 4.2 in Section 3.5 listed the Java classes that could be generated from a service’s WSDL document. Notice that two Java classes are generated for each service clause. This is because the generated service interface is unable to instantiate a client stub directly. Some generated client-side binding classes are later incorporated into JACK Teams plans to be used by their respective proxy agents, while others such as the port type generated Java interfaces are parsed to create events and roles for their respective teams.

<table>
<thead>
<tr>
<th>SNO</th>
<th>WSDL clauses</th>
<th>Java class(es) generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For each Type</td>
<td>A Java class</td>
</tr>
<tr>
<td>2</td>
<td>For each PortType</td>
<td>A Java interface</td>
</tr>
<tr>
<td>3</td>
<td>For each Binding</td>
<td>A stub class</td>
</tr>
<tr>
<td>4</td>
<td>For each Service</td>
<td>A service interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A service implementation (locator)</td>
</tr>
</tbody>
</table>

Table 4.2: WSDL2Java client-side bindings conversions [2]

4.3 Developing JACK Teams codes

In order to take advantages of the default failure recovery mechanism in JACK Teams, we must be able to represent composite Web services using JACK Teams programming constructs. Although an interactive tool and a specific parser module can be used to automatically generate these required JACK Teams codes, but the interactive tool and the parser module are not the main objectives for this research project. Hence all JACK Teams codes are created manually in the JACK Development Environment (JDE). JDE is a cross-platform graphical editor suite written in Java for developing JACK agent and team based applications. Its drag-and-drop feature as well as its context-sensitive menus can assist the users in constructing JACK Teams codes.

The development of the JACK Teams codes must be done using a bottom-up approach. Hence, events are the first construct to be developed as an event is considered to be the lowest level construct among other JACK Teams concepts. This is followed by roles, then plans and teamplans, and teams must be created last. Section 3.4.1 mentioned that events and roles are conceptually similar to operations and port types offered by a Web service. Therefore events and roles can be created by parsing the Java interface generated earlier by the WSDL2Java tool. The events take the name of the method signatures declared in the Java interface, whereas the roles take the name of the generated Java interfaces. Furthermore, the events must extend the BDIMessageEvent class and include the
following statement: `#set recover behavior repost`. This enables the team entity to repost the same BDI event when the chosen plans or teamplans fail to handle it and the next plan gets selected for execution. Hence if team redundancy is to be supported, duplicate teamplans must be declared as the failed teamplan is discarded and it will not be available for selection.

Message exchanges between services and their respective teams are bound to take place during the invocation of the component services and these messages could be of complex types. Hence the Java class generated earlier by the WSDL2Java tool should be incorporated in events to store these complex message types. For each event, there must exist at least one plan in the team that are capable of performing a role that handles the event. Events basically does nothing but store the messages passing between the containing team and its sub-teams. Plans must be created according to how the consumer wanted the events to be triggered. Plans must incorporate proxies in order to invoke their respective Web services and this can be achieved by instantiating the stub classes generated earlier. Furthermore, plans cannot be accessed by teamplans that coordinate the triggering of events. Therefore plans must store the results of invoked Web service operations back into their respective events and the teamplans must keep a local copy of these events in order to obtain the return results.

Teamplan mainly contain methods to choose sub-teams to perform the required roles and to describe how these sub-teams should cooperate to perform the tasks required to achieve the containing team’s overall goals. Sub-teams are selected from their role containers that are attached to the containing team when `#requires role` statements are declared in a team. After creating all other JACK Teams constructs as described earlier, the teams are developed at last. As a team represents specific Web service involved in the composite service and mainly contains declarations of plans or teamplans it uses and the roles it can perform, hence teams must be created after all other JACK constructs. The overall generation of JACK Teams representation of a composite service is illustrated in Figure 4.1.

Besides JACK Teams constructs, an external Java class that starts the JACK Teams application and a definition file that describes the team structure of the composite service must be created as well. Although these files can be compiled and executed together with the JACK Teams code in JDE, they cannot be developed using the JDE. Hence common editors like WordPad and Notepad are used instead.
4.4 Implementing the economical failure recovery

Section 3.5 described the necessity of an external data structure to monitor the execution status of each step in a teamplan. The following list describes the fields required by the data structure and their uses:

- **stepNo**: Integer. uniquely identifies each step,
- **roleRef**: String. indicates the role reference responsible for a specific step,
- **executionStatus**: String. indicates either NOT_EXECUTED or SUCCESS,
- **dependencies**: Integer array. indicates the steps whose output is being depended upon.

Section 4.3 mentioned that duplicate teamplans are required to support team redundancy. As a result, two possible scenarios may occur. In the first situation, a duplicate teamplan is activated and information stored within the data structure is still applicable to the current teamplan execution. In the second situation where a different teamplan is selected instead, information of the previous teamplan execution no longer applies and the current teamplan needs to create its own data structure. Since failed teamplans are no longer accessible to its calling team, the data structure must be kept in the containing team but at the same
time remains accessible to all its teamplans. In addition to the data structure, the containing team must contain a variable to keep track of the teamplan currently selected for execution. The teamplans must also contain a constant to uniquely identifies itself so that the containing team can compare if duplicate teamplans are selected.

Whenever a teamplan fails, its reasoning method `fail()` is triggered to perform post-processing for the failed plan. Therefore the `fail()` method in teamplan is the ideal location to implement the failure recovery algorithm described in Section 3.5. The `stepNo` attribute in the data structure identifies the steps in a teamplan from value “1” onwards as the number “0” is reserved to represent no dependencies for a specific step. The `executionStatus` attribute for each data structure must be checked before each step is executed. The execution of each step proceeds if its `executionStatus` is stated as `NOT_EXECUTED` and is skipped, if otherwise. The `executionStatus` is then changed to `SUCCESS` after a step is successfully executed. As each data structure instance only captures information for a single step, it is then necessary to keep a vector of data structures since there is no fixed number of steps in each teamplan.

Each teamplan starts off by checking if it is a duplicate of the previous teamplan. The teamplan either ends up constructing its own copy of data structure using information of its own steps or populates its data structure with information from the previous teamplan. When a teamplan fails, the failure algorithm of reasoning method `fail()` first checks which is the step that causes the failure. This is done by tracing the `executionStatus` from the first step to the last step in the vector as the first step being encountered with `NOT_EXECUTED` status is the cause of failure. We then trace the `roleRef` variable and the `dependencies` array from the first step to the failed step to check which are the steps that are executed by the same role filler and which are the steps that depends on the result of the failed step or vice versa. Steps that belong to either category will have their `executionStatus` changed back to `NOT_EXECUTED`. At the end of the algorithm, the teamplan writes its data structure back into the containing team to be used for the next chosen teamplan. The next section discusses some examples of the implemented Java code for the data structure as well as the JACK Teams representation of a travel agency composite service.

4.5 Case study

In this section, we use the same travel agency composite service as described in Figure 3.2 to explain how economical failure recovery for composite services can be achieved in JACK Teams. Meanwhile, error recovery at individual level, at the
team entity level, as well as making use of redundant team members to achieve robustness through replication are all implemented and tested. The objective of this travel agency composite service is to integrate four atomic services to achieve the following goals:

- reserve a flight ticket,
- reserve a hotel room,
- and rent a car from a depot that is nearby the reserved hotel.

In order to achieve the above goals, we must implement a JACK Teams program according to the description in Section 4.3. Appendix B lists the entire JACK Teams program that represents the travel agency composite service. In the following subsections, we shall discuss about the JACK Teams implementation and the data structure that contribute to the economical failure recovery behaviour in JACK Teams.

4.5.1 Invocation of Web services

Appendix B.1.3 shows how the implemented JACK Teams program can be used to invoke Web services through ReservingEconomic plan. In its reasoning method body(), ReservingEconomic instantiates a flight reservation Web service’s stub class and subsequently made use of this stub instance to invoke an operation in the flight reservation service. Referring to Section 3.5, notice that both the event and the invoked method bear the same name. This is to avoid discrepancy between the codes if a separate parser module is built in the future. Output from the service invocation must be written back into the event in all plans. This is because the teamplans that trigger these events are unable to access data from other plans. Return results can only be obtained if teamplans keep a local copy of the triggered events and all selected plans must explicitly record the results back into the respective events.

4.5.2 Persistency at individual level and the containing team level

In appendix B.1.1 and B.1.4, the ReserveFlight event and the Flight team codes are examples in our implemented JACK Teams program to achieve persistency at the individual level. Referring to the ReserveFlight event, we can see that it extends the BDIMessageEvent and contains a statement
#set behavior Recover repost. The set statement informs the JACK execution engine to repost the ReserveFlight event to the Flight team whenever a chosen plan fails to handle it. A plan failure will also cause an exception to be thrown and this exception will be caught in its reasoning method fail() to perform the required post-processing steps. When the Flight team receives the ReserveFlight event, it chooses another plan to handle it. The Flight team only fails when all of its plans failed to handle ReserveFlight. This will cause the teamplan that delegates the task to the Flight team to fail as well.

Persistency at the containing team level is the same as how persistency at the individual team member level is achieved. The only difference is that teamplans instead of plans are used at the team level. In Appendix B.5.1, B.5.2, and B.5.3, we implemented the goal-oriented behaviour in the containing team, User. The Composition event demonstrated similar characteristics as the ReserveFlight event such as extending the BDIMessageEvent and containing the #set behavior Recover repost statement. Like Flight team in Section 4.5.2, the User team also declares additional teamplans to handle the Composition event. However, unlike Flight team, User provides higher-level reasoning methods that perform team establishment and delegation of task to its team members.

4.5.3 Redundancy of teams

structure.def, Main.java, and ServiceComposition1 in appendix B.6.1, B.6.2, and B.5.2 show the codes that demonstrate how JACK Teams can achieve the redundancy of teams. structure.def is an initialisation file in JACOB format [4] that help the JACK Execution Engine to identify the role fillers for a specific JACK Teams program. Main.java is a normal Java program that instantiates the required teams in the JACK Teams program and triggers the event for the containing team to handle. It is important to note that the teams instantiated in Main.java must correspond to the teams initiated by structure.def. Otherwise, the JACK Teams program may either fail because less teams are instantiated or it may execute without choosing teams that are not initiated. The establish() reasoning method in ServiceComposition1 establishes the task team by choosing ACTIVE teams to fulfill its goals. In order to provide team redundancy, we must set the state of the failed team to be INACTIVE in the fail() reasoning method so that the same failed team will not be available for selection when the next teamplan is executed. As mentioned in Section 3.5, duplicate teamplans are required to support team redundancy. Hence a duplicate teamplan is created as ServiceComposition2, but we decided not to include it in the appendix since it is exactly the same as ServiceComposition1.
4.5.4 Economical failure recovery

Appendix B.6.3, B.5.2, and B.5.3 display how economical failure recovery can be provided in a JACK Teams program through DataStructure.java, ServiceComposition1.teamplan, and User.team. DataStructure.java provides a means to keep track of the execution status and the dependencies for each step in ServiceComposition1. ServiceComposition1 begins by establishing its task team to fulfill the above goals described earlier. This is followed by executing the statements in the reasoning method body(). Of which, the first statement calls for a helper method initialiseDS(), which checks if the current teamplan selected is a duplicate by comparing the teamplan number. If it is a duplicate, ServiceComposition1 will load the data structure stored in User. Otherwise, ServiceComposition1 will create its own data structure. The JACK execution engine then proceeds to execute the remaining statements in body() and this includes the steps required to achieve the aforementioned goals. Notice that the executionStatus is checked before each step is executed, this is to prevent successfully-executed steps of a previous teamplan from being re-executed.

When ServiceComposition1 fails, the fail() reasoning method will be triggered. This fail() reasoning method contains a failure recovery algorithm discussed in Section 3.5. The algorithm first checks which of the steps in ServiceComposition1 failed, and change the executionStatus of the failed step to NOT_EXECUTED. At the same time, the algorithm discovers which team is responsible to handle the failed step. It then goes through the set of data structures to find out which of these steps are performed by the failed team and change their executionStatus to NOT_EXECUTED. The algorithm then checks for dependencies of each data structure to find out if its corresponding step depends on the result of any step performed by the failed team. The executionStatus of these steps are then changed to NOT_EXECUTED. Lastly, ServiceComposition1 will write its copy of data structures back into User team where it may be used by the next chosen teamplan if it is a duplicate.
CHAPTER 5

Experimentation and results

Using a travel agency composite service as our case study, the program comprises of event codes, plan codes, role codes, and team codes for each component service and for the composite service as well. Other required codes for this program is a definition file that defines the available role fillers for the team establishment phase, a main Java program that constructs instances of the team representing each Web service, and a Java class that defines the attributes and methods of a data structure to support the proposed failure recovery algorithm.

The objective of this program is to enable a user to reserve a flight ticket from a flight reservation service, book a room from a hotel accommodation reservation service, and to rent a vehicle from a car rental service. The sequence in which these services are being invoked follows exactly to the order described in Figure 3.2.

A total of five experiments are designed to test our implementation on the different aspects of failure recovery and the test cases and results are described in the following sections:

5.1 Test cases

Our implementation will be tested using JACK Development Environment (JDE). In order to carried out the following test cases, sample Web services are created using Apache Axis 1.1 and deployed on Apache Tomcat Server 5.5. The test cases include:

1. Test the plans in the implementation to invoke Web services and stores the results of these services back into the event that each plan handles.

2. Test the sub-teams in the implementation to persist in achieving their specific goals. The sub-team to be tested is Flight.team.
3. Test the team entity in the implementation to persist in achieving the overall team goals. The team entity in this case is User.team.

4. Test the teamplans in the implementation to perform swapping of team members after the failed member had tried all ways to achieve its goals.

5. Test a duplicate teamplan in the implementation to perform swapping of team members without re-executing successful steps.

5.2 Expected and actual results

The expected results for all the test cases are shown in Figure 5.1.

<table>
<thead>
<tr>
<th>Case</th>
<th>Test case description</th>
<th>Expected results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ReserveFirst plan, ReserveBusiness plan, and ReserveEconomic plan are executed to handle ReserveFlight event.</td>
<td>Screen prompts to display the results of service invocations with either valid results or error messages.</td>
</tr>
<tr>
<td>2</td>
<td>Same as test case (1).</td>
<td>Screen prompts to display only the result of ReserveEconomic plan and the error messages of ReserveBusiness plan.</td>
</tr>
<tr>
<td>3</td>
<td>FailsTeamPlan1,teamplan, ServiceComposition1,teamplan, and ServiceComposition2,teamplan are executed to handle Composition.event.</td>
<td>Screen prompts to display error messages of FailedTeamPlan1,teamplan and ServiceComposition1,teamplan, and the success for successful execution of ServiceComposition2,teamplan.</td>
</tr>
<tr>
<td>4</td>
<td>CarA.team and CarB.team are triggered to execute their plans to handle BookCar.event.</td>
<td>Screen prompts to display error message of CarB.team which is CarA.team which in this case is CarB.team and the result of CarBooking plan in CarA.team which in this case is CarB.booking.</td>
</tr>
<tr>
<td>5</td>
<td>ServiceComposition1,teamplan and ServiceComposition2,teamplan are executed to handle Composition.event.</td>
<td>Screen prompts to display the error message for one of the steps in ServiceComposition1,teamplan and the results of all other steps except step 1 since step 1 is an independent step in ServiceComposition1,teamplan and should not be re-executed.</td>
</tr>
</tbody>
</table>

Figure 5.1: Expected results for each test case.

The actual result for test case (1) and (2) shows both the invocation of Web services and the persistence in achieving its goals at a sub-team level. The Flight team first selects ReservingFirst plan to handle the ReserveFlight event. After ReservingFirst fails, ReservingBusiness plan is chosen for execution next. However, ReservingBusiness fails as well. Hence the Flight team selects ReservingEconomic plan. ReservingEconomic successfully invokes the flight reservation service and the resulting reservation number is displayed in Figure 5.2.
The actual result for test case (3) shows the swapping from \texttt{carB.team} to \texttt{carA.team} after \texttt{carB.team} tries all its plans to handle the \texttt{BookCar} event and fail. As described in Section 3.5, swapping of team members is not a trivial matter as new task team is being established and the steps that are successfully-executed and do not depend on the output of other steps will not be re-executed when a duplicate teamplan is loaded. The actual result for test case (3), (4), and (5) are shown in Figure 5.3 and 5.4.

5.3 Comparison of actual and expected results

From our results, we have shown that the implementation is capable of satisfying all the default failure recovery mechanism in JACK Teams for composite services. Most importantly from the results, we showed that the implementation also supports the economical failure recovery proposed in Section 3.5. Hence we have
verified the implementation and it is considered as a success to provide efficient failure recovery for composite services using BDI Team-oriented agent structure.
CHAPTER 6

Conclusion and Future work

In this chapter, we conclude our work and discuss the open problems of our research and some future work that can be done to improve the creation of composite services as JACK Teams programs. We also raised the possibility of reusing JACK Teams programs as normal Web services for future work.

6.1 Conclusion

In this research project, we discussed why failures in Web services are inevitable. We also stressed the importance of non-functional attributes in Web services such as reliability and availability. Considering that without availability, even the most useful Web service would be meaningless to the service consumers. The factors that may cause failures in Web services can be grouped into three main categories: Hardware failures, Software failures, and Human failures.

Since a composite Web service is about coordinating the collaboration from various business organisations, be it inter-enterprise or intra-enterprise, we presented a view of service composition as a loose form of teamwork. We primarily focus on the failure recovery aspects of an agent-oriented team architecture, named JACK Teams. In this dissertation, we discussed how Web services can be incorporated into JACK Teams and how JACK execution engine can be used to execute composite services.

Currently, failure recovery for composite services are mainly provided by two error recovery techniques: backward error recovery, or forward error recovery. We provided a discussion on both recovery approaches and gathered properties from WSCA and WSTx standards. The fault-tolerance support in JACK Teams is then evaluated using these properties and JACK Teams seems to be suitable in providing failure for composite services. We then proposed to use the default failure recovery mechanism in JACK Teams for composite services creation and execution. Our main contribution to the field of adaptive system is in describing
and implementing a more economical and efficient way of recovering from Web service failures using JACK Teams.

At the end of the research project, the implementation incorporates the persistence to achieve the composite service’s goals at both the individual level and the service composition level. The implementation also took advantage of the replication and redundancy of Web services by enabling failed team members to be replaced by others that are capable of performing the same task. Lastly, we managed to incorporate a form of data structure into the JACK Teams program to keep track of the execution status and the dependencies of results from other steps’ output for each step. This enables successfully-executable steps to be retained if a teamplan fails and contributes to better efficiency in executing composite services using JACK Teams.

6.2 Future works

Although we have managed to create and execute service composition using JACK Teams, the resulting composite services are not reusable by other service consumers. One way of overcoming this problem is to expose the resulting JACK Teams programs into BPEL or WSDL documents so that the JACK Teams code can be reused just like any other normal Web services.

Currently, JACK Teams is our choice of BDI structure being employed to provide failure recovery for composite Web services. However, there are other BDI agent architectures, such as Jadex which does not support teamwork, that can be used to generate more general forward error recovery techniques.

As the emphasis of this research is on the failure recovery portion, the translation of Web services into JACK Teams concepts and development of the teamplans are done manually via JDE. Another opportunity for future work is to implement a specific parser module to support automatic translation of Java classes into JACK Teams codes. This is clearly something that can be done as we have shown in our implementation and discussion how these classes can be mapped into JACK Teams programming constructs.
Original Honours Proposal

Background

Due to the heterogeneous nature of Web services, it is sometimes difficult to find the appropriate service to solve the users’ problems at hand. This is especially so when it comes to accomplishing a much complex task. For example, booking tour packages where one may need to search for the cheapest flight, book accommodation, find attractions, and hire a car. Although there are services available to perform each individual task but there is none to perform them all. Another example of a common complex task is when the new occupants move into an apartment, they will require a single integration of services to allow them to subscribe to utility, telephone, gas, and Internet services from different service providers. It is very difficult to achieve these general tasks by a single service. Therefore one way to overcome this difficulty is to compose relevant services together to satisfy the different needs of the users.

At present, Web technologies has evolved to provide us with the description, and a platform-independent and standardized communication mechanism for Web services. There are also standards to describe service composition, but there is no mechanism available to coordinate this integrated execution. One way of viewing service composition is as a form of teamwork[19]. The structure of a team is generally made up of different individuals. For example, a soccer team can have twenty two members including reserves where each has his/her own beliefs and attributes. However, all the members share a common goal, which is to execute all the tasks that are required of them according to a team plan in order to win every match. These teams usually have a team manager to coordinate its activities such as selecting eleven members in the team to play a match. The reserves in the team act as a form of backup to replace any members injured during a match. We can view a match as a user’s service request, the eleven members as the services involved in a composition plan, and the reserve as services that can perform the same tasks required in a composition plan. Therefore implementing
the team concept enables the failure recovery mechanism to be provided for service composition. Such property is highly desirable particularly as Web services operate in an open system environment, which has a high failure rate.

In this research project, I will be looking at composite Web service execution using a team-oriented approach and to implement such a system using BPEL4WS for representation, JACK Teams for execution, and lastly using WSDL to republish the composite services so that it can be reuse like any other atomic services.

Aim

The main aim of this research project is to implement a Web service composition system using JACK Intelligent Agents Teams as the basis and subsequently carry out a few case studies on openNet and Australian Tourism Data Warehouse (ATDW). However, a specific case study on the implementation would be to use real data provided by ATDW as it will provide more realistic feedback.

The following are what I expect to accomplish by undertaking this project:

- Implementation of a Web service composition system using JACK Teams, Business Process Execution Language 4 Web Services (BPEL4WS) and WSDL.
- Develop case study using real data from the Australia Tourism Data Warehouse (ATDW) and Open Net to test the implemented system on different levels of failure recovery.
- Prove our hypothesis is correct not only for specific case but also general cases that team oriented model is suitable for Web service composition.

Method

Padgham and Liu [19] have done some relevant research on why service composition should be viewed as a loose form of teamwork. It was mentioned in their findings [19] that the advantages of using team-oriented model software such as JACK Teams for Web service composition are its ability to develop team plans to coordinate the integrated execution of services, its ability to execute these team plans with Web services as Role Fillers and its robustness. It was also mentioned that in the event of a failure of an individual service, the execution of a chosen team plan will not be given up until all other replacement services are tested and
failed as well. This is why failure recovery of JACK Teams is very important because we can never be certain that a service in the open environment previously identified or used is still available. Hence the assurance that JACK Teams will only give up a team plan after trying all other replacement services to fill the failing role, together with the commitment of JACK Teams to try out different plans if one fails provide a higher chance of achieving a goal.

Harkey et. al. [11] mentioned that WSDL allows Web services to describe a service’s interfaces and bindings. Before WSDL was introduced, the developers tried implementing SOAP-based Web services. Although SOAP provided means of communication between the service requester and the service provider, it was not robust enough to perform type checking for the parameters. Therefore, WSDL was incorporated to allow Web services to be self-describing. WSDL comprises various components such as operation, port type, binding, and partnerLinkType. Operation is an abstract definition of an operation for a message such as a business process that will accept, process a message, and send the response back as a message. For example, a student Z tries to book his/her lab for a particular unit; Z sends a message to the timetable server, which will process the message, and subsequently send a message back to Z notifying if Z managed to grab the desired slot. Port type, on the other hand, defines a set of operations implemented by some service providers who provides a common service whereas Binding defines how an operation is bound to a protocol. A partnerLinkType specifies possible collaborations with other services as partners and it is used in BPEL4WS for partnership verification. A partnerLinkType describes its partner links from a central controller’s point of view and it specifies the capabilities of each partner as partnerRole and itself as myRole.

As mentioned earlier, partnerLinkType is used in BPEL4WS for partnership verification when parsing the WSDL document during a BPEL4WS process. During the process, the operations component in WSDL which maps to the JACK events, partnerRole maps to the JACK Role type, and myRole maps to the Team entity itself, resulted in team members, events, and roles to be generated. The process steps in BPEL4WS make up the parts of an interaction (receive, reply, and invoke). The flow controls of BPEL4WS then organize the interaction into sequential, parallel, repetitive, selection and triggered selection using sequence, flow, while, switch, and pick. Sequencing, looping and selection in JACK teams use standard Java. Padgham and Liu [19] state that parallelism in JACK teams can be implemented using @parallel() and JACK Teams’ @wait_for() can be used to trigger an operation after a certain action is detected or a timer times out. One of the main motivations for translating BPEL4WS specifications into JACK Teams is to be able to integrate composite services into a larger application, which supports agent reasoning.
The following outline the tasks which I wish to achieve by undertaking this project:

1. Discuss about WSDL and its messaging protocols.

2. Discuss about BPEL4WS and how it provides different forms of execution (sequential, repetitive, selection and parallel).

3. Discuss about team-oriented model software such as JACK Teams and STEAM; their differences and why JACK Teams is more suitable for this project.

4. Elaborate on the failure recovery mechanism, executing team plans with Web services as Role Fillers principle of JACK Teams.

5. Implementing the Web service composition system and developing test cases to evaluate it.

6. Discuss about SHOP2 and how HTN planners can be incorporated as part of the Web service composition system.

Tasks

<table>
<thead>
<tr>
<th>SNO</th>
<th>SUB TASK</th>
<th>ETA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research proposal submission</td>
<td>17 March</td>
</tr>
<tr>
<td>2</td>
<td>Reading up WSDL’s W3C Note</td>
<td>24 March</td>
</tr>
<tr>
<td>3</td>
<td>Reading up specification of BPEL4WS</td>
<td>4 April</td>
</tr>
<tr>
<td>4</td>
<td>Reading up JACK Teams Manual</td>
<td>14 April</td>
</tr>
<tr>
<td>5</td>
<td>Literature review</td>
<td>28 April</td>
</tr>
<tr>
<td>6</td>
<td>Developing case studies and design</td>
<td>19 May</td>
</tr>
<tr>
<td>7</td>
<td>Revised research proposal submission</td>
<td>26 May</td>
</tr>
<tr>
<td>8</td>
<td>Coding</td>
<td>25 June</td>
</tr>
<tr>
<td>9</td>
<td>System testing</td>
<td>16 July</td>
</tr>
<tr>
<td>10</td>
<td>System evaluation</td>
<td>23 July</td>
</tr>
<tr>
<td>11</td>
<td>Drafting of thesis</td>
<td>21 August</td>
</tr>
<tr>
<td>12</td>
<td>Designing of poster</td>
<td>28 August</td>
</tr>
<tr>
<td>13</td>
<td>Correction of draft thesis</td>
<td>5 September</td>
</tr>
<tr>
<td>14</td>
<td>Draft thesis submission</td>
<td>14 September</td>
</tr>
<tr>
<td>15</td>
<td>Final thesis submission</td>
<td>20 October</td>
</tr>
</tbody>
</table>
B.1 Flight Reservation Web service codes

B.1.1 Events.

```java
public event ReserveFlight extends BDIMessageEvent {
    public java.lang.String flightDate;
    public java.lang.String location;
    public java.lang.String budget;
    public java.lang.String flightClass;
    public java.lang.String result;

    #posted as
    EvMethod1(java.lang.String flightDate, java.lang.String location,
              java.lang.String budget, java.lang.String flightClass) {
        this.flightDate = flightDate;
        this.location = location;
        this.budget = budget;
        this.flightClass = flightClass;
    }

    #set behavior Recover repost;

    public void setResult(java.lang.String result) {
        this.result = result;
    }

    public java.lang.String getResult() {
        return result;
    }

    public void setFlightClass(java.lang.String flightClass) {
        this.flightClass = flightClass;
    }
}
```

B.1.2 Role

```java
public role FlightResWS extends Role {
    #handles event CancelFlight ce;
    #handles event ConfirmFlight cn;
    #handles event GetAirportLoc gt;
    #handles event ReserveFlight re;
}
```
B.1.3 Plans

public plan ReservingFirst extends Plan {
  #handles event ReserveFlight re;
  
  static boolean relevant(ReserveFlight ev) {
    return true;
  }
  context() {
    true;
  }
  #reasoning method body() {
    System.out.println("\tI am " + getAgent().name());
    try {
      flight.FlightResWS stub = (new flight.FlightResWSServiceLocator()
          .getFlightResWS());
      re.setFlightClass("First");
      String temp = stub.ReserveFlight(re.flightDate, re.location,
          re.budget, re.flightClass);
      if (temp == null)
        false;
      re.setResult(temp);
    } catch (Exception ex) {ex.printStackTrace();}
  }
  #reasoning method fail() {
    System.out.println("\tPlan Reserving" + re.flightClass + " fails.\n");
  }
}

class ReservingBusiness extends ReservingFirst {
  #handles event ReserveFlight re;
  
  static boolean relevant(ReserveFlight ev) {
    return true;
  }
  context() {
    true;
  }
  #reasoning method body() {
    System.out.println("\tI am " + getAgent().name());
    try {
      flight.FlightResWS stub = (new flight.FlightResWSServiceLocator()
          .getFlightResWS());
      re.setFlightClass("Business");
      String temp = stub.ReserveFlight(re.flightDate, re.location,
          re.budget, re.flightClass);
      if (temp == null)
        false;
      re.setResult(temp);
    } catch (Exception ex) {ex.printStackTrace();}
  }
  #reasoning method fail() {
    System.out.println("\tPlan Reserving" + re.flightClass + " fails.\n");
  }
}

class ReservingEconomic extends ReservingFirst {
  #handles event ReserveFlight re;
  
  static boolean relevant(ReserveFlight ev) {
    return true;
  }
}
context() {
    true;
}

#reasoning method body() {
    System.out.println("\tI am " + getAgent().name());
    try{
        flight.FlightResWS stub = (new flight.FlightResWSServiceLocator()
          ).getFlightResWS();
        re.setFlightClass("Economic");
        String temp = stub.ReserveFlight(re.flightDate, re.location,
          re.budget, re.flightClass);
        re.setResult(temp);
    }catch(Exception ex){ex.printStackTrace();}
}

#reasoning method fail() {
    System.out.println("\tPlan Reserving"+re.flightClass+" fails.
"+"\n");
}

B.1.4 Team

public team Flight extends Team {
    #uses plan CancellingFlight;
    #uses plan ConfirmingFlight;
    #uses plan GettingAirport;
    #uses plan ReservingFirst;
    #uses plan ReservingBusiness;
    #uses plan ReservingEconomic;
    #performs role FlightResWS;

    public Flight(String name) {
        super(name);
    }
}

B.2 Car Rental Web service codes

B.2.1 Events

public event GetDepotLoc extends BDIMessageEvent {
    public java.lang.String result;

    #posted as EvMethod1() {}
    #set behavior Recover repost;

    public void setResult(java.lang.String result) {
        this.result = result;
    }
    public java.lang.String getResult(){
        return result;
    }
}
public event CarBooking extends BDIMessageEvent {
    public java.lang.String dateFrom;
    public java.lang.String duration;
    public java.lang.String brandModel;
    public java.lang.String result;

    // posted as EvMethod1(java.lang.String dateFrom, java.lang.String duration, java.lang.String brandModel) {
    //     this.dateFrom = dateFrom;
    //     this.duration = duration;
    //     this.brandModel = brandModel;
    // }

    // set behavior Recover repost;
    public void setResult(java.lang.String result) {
        this.result = result;
    }
    public java.lang.String getResult(){
        return result;
    }
}

B.2.2 Role

public role CarRentalWS extends Role {
    #handles event CancelBooking ca;
    #handles event ConfirmBooking co;
    #handles event GetDepotLoc ge;
    #handles event CarBooking cr;
}

B.2.3 Plans

public plan BookingCar extends Plan {
    #handles event CarBooking cr;
    static boolean relevant(CarBooking ev) {
        return true;
    }
    context() {
        true;
    }
    // reasoning method body() {
    //     System.out.println("I am " + getAgent().name());
    //     try{
    //         car.CarRentalWS stub = (new car.CarRentalWSServiceLocator ()).getCarRentalWS();
    //         String temp = stub.CarBooking(cr.dateFrom, cr.duration, cr.brandModel);
    //         cr.setResult(temp);
    //     }catch(Exception ex){ex.printStackTrace();}
    // }
}

public plan GettingDepot extends Plan {
    #handles event GetDepotLoc ge;
    static boolean relevant(GetDepotLoc ev){

return true;
}
context() {
    true;
}

#reasoning method body() {
    System.out.println("I am " + getAgent().name());
    try{
        car.CarRentalWS stub = (new car.CarRentalWSServiceLocator())
            .getCarRentalWS();
        String temp = stub.GetDepotLoc();
        ge.setResult(temp);
    }catch(Exception ex){ex.printStackTrace();}
}

B.2.4 Team

public team CarA extends Team {
    #uses plan CancellingBooking;
    #uses plan ConfirmingBooking;
    #uses plan GettingDepot;
    #uses plan BookingCar;
    #performs role CarRentalWS;

    public CarA(String name) {
        super(name);
    }
}

B.3 Hotel Accommodation Reservation Web service codes

B.3.1 Events

public event GetHotelLocation extends BDIMessageEvent {
    public java.lang.String resvNo;
    public java.lang.String result;

    #posted as EvMethod1(java.lang.String resvNo) {
      this.resvNo = resvNo;
    }
    #set behavior Recover repost;
    public void setResult(java.lang.String result) {
      this.result = result;
    }
    public java.lang.String getResult(){
      return result;
    }
}

public event ReserveRoom extends BDIMessageEvent {
    public java.lang.String dateFrom;
    public java.lang.String duration;
public java.lang.String budget;
public java.lang.String result;

#posted as EvMethod1(java.lang.String dateFrom, java.lang.String duration,
java.lang.String budget) {
    this.dateFrom = dateFrom;
    this.duration = duration;
    this.budget = budget;
}
#set behavior Recover repost;
public void setResult(java.lang.String result) {
    this.result = result;
}
public java.lang.String getResult(){
    return result;
}

public event ConfirmRoom extends BDIMessageEvent {
    public java.lang.String resvNo;
    public boolean result;

    #posted as EvMethod1(String resvNo) {
        this.resvNo = resvNo;
    }
    #set behavior Recover repost;
    public void setResult(boolean result) {
        this.result = result;
    }
    public boolean getResult(){
        return result;
    }
}

B.3.2 Role

public role HotelResWS extends Role {
    #handles event CancelRoom cl;
    #handles event ConfirmRoom cf;
    #handles event GetHotelLocation gh;
    #handles event ReserveRoom rr;
}

B.3.3 Plans

public plan GettingHotel extends Plan {
    #handles event GetHotelLocation gh;
    static boolean relevant(GetHotelLocation ev) {
        return true;
    }
    context() {
        true;
    }
    #reasoning method body() {
        System.out.println("\tI am " + getAgent().name());
        try{
            hotel.HotelResWS stub = (new hotel.HotelResWSServiceLocator

getHotelResWS();
String temp = stub.GetHotelLocation(gh.resvNo);
gh.setResult(temp);
}catch(Exception ex){ex.printStackTrace();}
}
}

public plan ReservingRoom extends Plan {
#handles event ReserveRoom rr;
static boolean relevant(ReserveRoom ev) {
    return true;
}
context() {
    true;
}
#reasoning method body(){
    System.out.println("I am "+getAgent().name());
    try{
        hotel.HotelResWS stub = (new hotel.HotelResWSServiceLocator()
        ).getHotelResWS();
        String temp = stub.ReserveRoom(rr.dateFrom, rr.duration,
        rr.budget);
        rr.setResult(temp);
    }catch(Exception ex){ex.printStackTrace();}
}
}

public plan ConfirmingRoom extends Plan {
#handles event ConfirmRoom cf;
static boolean relevant(ConfirmRoom ev) {
    return true;
}
context() {
    true;
}
#reasoning method body(){
    System.out.println("I am "+getAgent().name());
    try{
        hotel.HotelResWS stub = (new hotel.HotelResWSServiceLocator()
        ).getHotelResWS();
        boolean temp = stub.ConfirmRoom(cf.resvNo);
        cf.setResult(temp);
    }catch(Exception ex){ex.printStackTrace();}
}
}

B.3.4 Team

public team Hotel extends Team {
#uses plan CancellingRoom;
#uses plan ConfirmingRoom;
#uses plan GettingHotel;
#uses plan ReservingRoom;
#performs role HotelResWS;
public Hotel(String name) {
    super(name);
}
}
B.4 Street Directory Web service codes

B.4.1 Events

public event IsNear extends BDIMessageEvent {
    public java.lang.String streetFrom;
    public java.lang.String streetTo;
    public boolean result;

    #posted as EvMethod1(String streetFrom, String streetTo) {
        this.streetFrom = streetFrom;
        this.streetTo = streetTo;
    }
    #set behavior Recover repost;
    public void setResult(boolean result) {
        this.result = result;
    }
    public boolean getResult() {
        return result;
    }
}

B.4.2 Role

public role StreetDirWS extends Role {
    #handles event CheckDistance cd;
    #handles event IsNear is;
}

B.4.3 Plans

public plan ProximityChecking extends Plan {
    #handles event IsNear is;

    static boolean relevant(IsNear ev) {
        return true;
    }
    context() {
        true;
    }
    #reasoning method body() {
        System.out.println("I am " + getAgent().name());
        try {
            street.StreetDirWS stub = (new street.StreetDirWSServiceLocator())
                .getStreetDirWS();
            boolean temp = stub.IsNear(is.streetFrom, is.streetTo);
            is.setResult(temp);
        } catch (Exception ex) { ex.printStackTrace(); }
    }
}
B.4.4 Team

public team Street extends Team {
    #uses plan DistanceChecking;
    #uses plan ProximityChecking;
    #performs role StreetDirWS;

    public Street(java.lang.String name) {
        super(name);
    }
}

B.5 Composite service codes

B.5.1 Event

public event Composition extends BDIMessageEvent {
    #posted as startExecution() {}
    #set behavior Recover repost;
}

B.5.2 Teamplan

public teamplan ServiceComposition1 extends TeamPlan {
    CancelBooking cancelBk;
    ConfirmBooking confirmBk;
    GetDepotLoc getDepot;
    CarBooking carBk;
    CancelFlight cancelFlt;
    ConfirmFlight confirmFlt;
    GetAirportLoc getAirport;
    ReserveFlight reserveFlt;
    CancelRoom cancelRm;
    ConfirmRoom confirmRm;
    GetHotelLocation getHotel;
    ReserveRoom reserveRm;
    IsNear iNear;
    CheckDistance checkDist;
    static int tplanNo = 2;
    static String FLIGHTWS = "flt";
    static String CARWS = "car";
    static String HOTELWS = "hotel";
    static String STREETWS = "strt";
    User teamUser;
    Vector DS;

    aos.team.Role pickRole(java.util.Vector busy, aos.team.RoleContainer rc) {
        for (Enumeration e = rc.tags(); e.hasMoreElements(); ) {
            Role r = rc.find((String) e.nextElement());
            if (r.state == Role.ACTIVE && !busy.contains(r.actor)) {
                busy.add(r.actor);
                return r;
            }
        }
    }
}
return null;
}

void initialiseDS() {
    DS = new Vector();
    if(teamUser.compareTeamPlans(tplanNo)) {
        int size = teamUser.getDSSize();
        for(int i=0; i<size; i++){
            DS.add(teamUser.getDS(i));
        }
    }
    else {
        // create own data structure for this teamplan
        int step = 1;
        DataStructure temp;
        //creating data structure for step 1
        temp = new DataStructure(FLIGHTWS,step);
        temp.setDep(temp.NO_DEP);
        DS.add(temp);
        step++;
        //creating data structure for step 2
        temp = new DataStructure(HOTELWS,step);
        temp.setDep(temp.NO_DEP);
        DS.add(temp);
        step++;
        //creating data structure for step 3
        temp = new DataStructure(HOTELWS,step);
        temp.setDep(2);
        DS.add(temp);
        step++;
        //creating data structure for step 4
        temp = new DataStructure(CARWS,step);
        temp.setDep(temp.NO_DEP);
        DS.add(temp);
        step++;
        //creating data structure for step 5
        temp = new DataStructure(STREETWS,step);
        temp.setDep(4,3,2);
        DS.add(temp);
        step++;
        //creating data structure for step 6
        temp = new DataStructure(CARWS,step);
        int[] tDepArray = {5,4,3,2};
        temp.setDep(tDepArray);
        DS.add(temp);
        step++;
        //creating data structure for step 7
        temp = new DataStructure(HOTELWS,step);
        int[] tDepArray2 = {6,5,4,3,2,1};
        temp.setDep(temp.NO_DEP);
        DS.add(temp);
    }
}

#handles event Composition cmp;
#uses role CarRentalWS cc as car;
#uses role FlightResWS fc as flt;
#uses role HotelResWS hc as hotel;
#uses role StreetDirWS sc as strt;
#uses interface Team team;

static boolean relevant(Composition ev) {
    return true;
}
context() {
    true;
}

#reasoning method body()

    teamUser = (User) team.getAgent();
    initialiseDS();
    System.out.println("Team established for composite service: " + team.name());
    System.out.println("Team selected to fill CarRentalWS = " + car.actor);
    System.out.println("Team selected to fill FlightResWS = " + flt.actor);
    System.out.println("Team selected to fill HotelResWS = " + hotel.actor);
    System.out.println("Team selected to fill StreetDirWS = " + strt.actor);

    int step = 0;
    DataStructure temp = (DataStructure) DS.elementAt(step);
    if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
        System.out.println("\nExecuting step " + step + ":");
        reserveFlt = flt.re.EvMethod1("17/02/2006", "perth", "200", "");
        @team_achieve(flt, reserveFlt);
        System.out.println("The flight reservation number is " + reserveFlt.getResult());
        temp.setExecutionStatus(temp.SUCCESS);
        DS.set(step,temp);
    }
    step++;
    temp = (DataStructure) DS.elementAt(step);
    if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
        System.out.println("\nExecuting step " + step + ":");
        reserveRm = hotel.rr.EvMethod1("17/02/2006","5 days","500");
        @team_achieve(hotel, reserveRm);
        System.out.println("The room reservation number is " + reserveRm.getResult());
        temp.setExecutionStatus(temp.SUCCESS);
        DS.set(step,temp);
    }
    step++;
    temp = (DataStructure) DS.elementAt(step);
    if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
        System.out.println("\nExecuting step " + step + ":");
        getHotel = hotel.gh.EvMethod1(reserveRm.getResult());
        @team_achieve(hotel, getHotel);
        System.out.println("The location of the reserved hotel accommodation is " + getHotel.getResult());
        temp.setExecutionStatus(temp.SUCCESS);
        DS.set(step,temp);
    }
    step++;
    temp = (DataStructure) DS.elementAt(step);
    if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
        System.out.println("\nExecuting step " + step + ":");
        getDepot = car.ge.EvMethod1();
        @team_achieve(car, getDepot);
        System.out.println("The location of the desired car rental company's depot is " + getDepot.getResult());
        temp.setExecutionStatus(temp.SUCCESS);
        DS.set(step,temp);
    }
    step++;
    temp = (DataStructure) DS.elementAt(step);
    if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
        System.out.println("\nExecuting step " + step + ":");
        iNear = strt.is.EvMethod1(getHotel.getResult(),getDepot.getResult());
        @team_achieve(strt, iNear);
    }
}
System.out.println("The result of the proximity check for "+
iNear.streetFrom and "+iNear.streetTo is "+iNear.getResult());
temp.setExecutionStatus(temp.SUCCESS);
DS.set(step,temp);
}
step++;
if(iNear.getResult()==true){
temp = (DataStructure) DS.elementAt(step);
if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
    System.out.println("Executing step "+temp.getStepNo());
    carBk = car.cr.EvMethod1("17/02/2006","3 days","Toyota");
    @team_achieve(car,carBk);
    System.out.println("The car rental booking number is "+
carBk.getResult());
temp.setExecutionStatus(temp.SUCCESS);
    DS.set(step,temp);
}
step++;
temp = (DataStructure) DS.elementAt(step);
if(temp.getExecutionStatus()==temp.NOT_EXECUTED){
    System.out.println("Executing step "+temp.getStepNo());
    confirmRm = hotel.cf.EvMethod1(reserveRm.getResult());
    @team_achieve(hotel,confirmRm);
    System.out.println("The confirmation of room is "+confirmRm.getResult());
temp.setExecutionStatus(temp.SUCCESS);
    DS.set(step,temp);
}
}
#reasoning method establish() {
    Vector busy = new Vector();
car = (CarRentalWS) pickRole(busy, cc);
car != null;
flt = (FlightResWS) pickRole(busy, fc);
flt != null;
hotel = (HotelResWS) pickRole(busy, hc);
hotel != null;
strt = (StreetDirWS) pickRole(busy, sc);
strt != null;
}
#reasoning method fail() {
    System.out.println("Entering post processing method fail() to clean up
the failed steps in ServiceComposition1.teamplan.");
    //executing the failure recovery algorithm
    DataStructure failedStep, temp;
car.setState(car.INACTIVE);
    int size = DS.size();
    //STEP1 - check which step causes the failure
    for(int i=0;i<size;i++){
        if(failedStep == (DataStructure) DS.elementAt(i)){
            i = size;
        }
    }
    System.out.println("Step "+failedStep.getStepNo()" failed.
    //STEP2 - Mark the steps with the same role reference as NOT_EXECUTED
    //STEP3 - Mark steps that depended on failed step as NOT_EXECUTED
    for(int i=0;i<failedStep.getStepNo();i++){
        temp = (DataStructure) DS.elementAt(i);
        if(temp.getRoleRef().equals(failedStep.getRoleRef())){
            temp.setExecutionStatus(temp.NOT_EXECUTED);
        }
    }
}
if (failedStep.checkDep(temp.getStepNo())){
    temp.setExecutionStatus(temp.NOT_EXECUTED);
}

DS.set(i,temp);

//writing the data structure back into the containing team
teamUser.initialiseDS();
for(int i=0;i<size;i++){
    teamUser.addDS((DataStructure) DS.elementAt(i));
}

//reasoning method pass()
    System.out.println("nThe composite service is executed successfully.");

B.5.3 Team

public team User extends Team {
    #handles event Composition;
    #posts event Composition cmp;
    #uses plan FailedTeamPlan;
    #uses plan ServiceComposition1;
    #uses plan ServiceComposition2;
    #requires role CarRentalWS cc(1,1);
    #requires role FlightResWS fc(1,1);
    #requires role HotelResWS hc(1,1);
    #requires role StreetDirWS sc(1,1);

    public User(String name) {
        super(name);
        dStructure = new Vector();
        teamplanNo = 0;
    }

    private Vector dStructure;
    private int teamplanNo;
    public void start() {
        postWhenFormed(cmp.startExecution());
    }
    public void addDS(DataStructure ds) {
        dStructure.add(ds);
    }
    public boolean compareTeamPlans (int tplanNo) {
        if(this.teamplanNo==tplanNo)
            return true;
        else {
            this.initialiseDS();
            this.setTeamplanNo(tplanNo);
            return false;
        }
    }
    public void initialiseDS () {
        if(!dStructure.isEmpty())
            dStructure.clear();
    }
    private void setTeamplanNo (int tplanNo) {
        this.teamplanNo = tplanNo;
    }
}
public DataStructure getDS (int index) {
    if(index<=this.getDSSize())
        return (DataStructure) dStructure.elementAt(index);
    else
        return null;
}
public int getDSSize() {
    return dStructure.size();
}

B.6 Other codes

B.6.1 Definition file

<Team :name "The Holiday Stop"
 :roles (  
 <Role :type "CarRentalWS" :name "cc"
 :fillers (  
 <Team :name "Europcar@%portal" >
 <Team :name "Action@%portal" >
 )
 >
 <Role :type "FlightResWS" :name "fc"
 :fillers (  
 <Team :name "Flight Centre@%portal" >
 )
 >
 <Role :type "HotelResWS" :name "hc"
 :fillers (  
 <Team :name "Hilton Hotel@%portal" >
 )
 >
 <Role :type "StreetDirWS" :name "sc"
 :fillers (  
 <Team :name "Msn Mappoints@%portal" >
 )
 >
 )

B.6.2 Main Java program

public class Main {
    public static void main(String[] args) {
        new CarA("Europcar");
        new CarB("Action");
        new Flight("Flight Centre");
        new Hotel("Hilton Hotel");
        new Street("Msn Mappoints");
        User userAgent = new User("The Holiday Stop");
        userAgent.start();
    }
}
B.6.3 Data structure

```java
public class DataStructure {
    private String roleRef;
    private int executionStatus;
    private int stepNo;
    private int[] dependencies;
    public static int NOT_EXECUTED = 0;
    public static int SUCCESS = 1;
    public static int NO_DEP = 0;

    DataStructure(String rRef, int stepNo) {
        this.setRoleRef(rRef);
        this.setExecutionStatus(NOT_EXECUTED);
        this.stepNo = stepNo;
    }
    public String getRoleRef() {
        return roleRef;
    }
    public void setRoleRef(String ref) {
        this.roleRef = ref;
    }
    public int getExecutionStatus() {
        return executionStatus;
    }
    public int getStepNo() {
        return stepNo;
    }
    public void setExecutionStatus(int state) {
        this.executionStatus = state;
    }
    public int[] getDep() {
        return dependencies;
    }
    public boolean checkDep(int stepNo) {
        for(int i=0; i<dependencies.length; i++) {
            if(dependencies[i] == stepNo)
                return true;
        }
        return false;
    }
    public void setDep(int[] dep) {
        if(dep.length>0)
            dependencies = new int[dep.length];
        else
            dependencies = new int[1];
        for(int i=0; i<dep.length; i++) {
            dependencies[i] = dep[i];
        }
    }
    public void setDep(int d1) {
        int[] x = {d1};
        this.setDep(x);
    }
    public void setDep(int d1, int d2) {
        int[] x = {d1, d2};
        this.setDep(x);
    }
    public void setDep(int d1, int d2, int d3) {
        int[] x = {d1, d2, d3};
        this.setDep(x);
    }
}
```
Bibliography


