UTILIZING THE BLACKBOARD PARADIGM TO IMPLEMENT A WORKFLOW ENGINE

Salman Noor

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering

Johannesburg, 2015

Declaration

I declare that this dissertation is my own unaided work. It is being submitted to the Degree of Master of Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

_____

Salman Noor

Signed this _____ day of _____ 20 _____

Abstract

Workflow management has evolved into a mature field with numerous workflow management systems with scores of features. These systems are designed to automate business processes of organisations. However, many of these workflow engines struggle to support complex workflows. There has been relatively little research into building a workflow engine utilizing the blackboard paradigm. The blackboard paradigm can be characterized as specialists interacting with and updating a centralized data structure, namely the blackboard, with partial and complete solutions. The opportunistic control innate to the blackboard paradigm can be leveraged to support the execution of complex workflows. Furthermore, the blackboard architecture can be seen to accommodate comprehensive workflow functionality. This research aims to verify whether or not the blackboard paradigm can be used to build a workflow engine. To validate this research, a prototype was designed and developed following stringent guidelines in order to remain true to the blackboard paradigm. Four main perspectives of workflow management namely the functional, behavioural, informational and operational aspects with their quality indicators and requirements were used to evaluate the prototype. This evaluation approach was chosen since it is universally applicable to any workflow engine and thereby provides a common platform on which the prototype can be judged and compared against other workflow engines. The two most important quality indicators are the level of support a workflow engine can provide for 20 main workflow patterns and 40 main data patterns. Test cases based on these patterns were developed and executed within the prototype to determine the level of support. It was found that the prototype supports 85% of all the workflow patterns and 72.5% of all the data patterns. This reveals some functional limitations in the prototype and improvement suggestions are given that can boost these scores to 95% and 90% for workflow and data patterns respectively. The nature of the blackboard paradigm only prevents support of only 5% and 10% of the workflow and data patterns respectively. The prototype is shown to substantially outperform most other workflow engines in the level of patterns support. Besides support for these patterns, other less important quality indicators provided by the main aspects of workflow management are also found to be present in the prototype. Given the above evidence, it is possible to conclude that a workflow engine can be successfully built utilizing the blackboard paradigm.

Acknowledgements

I would like to sincerely thank my supervisor Stephen Levitt for his unwavering guidance and support throughout the course of this research. I would also like to extend my thanks to some members of the CNS division of Wits University namely Luci Carosin, Armonde Furmie, Chris Baker and Aandil Silal for their support and guidance during the design, development and testing of the prototye system contributing to this research.

I would like to thank my parents and other family members who have always supported me in all of life's endeavours.

Table of Contents

1 INTRODUCTION 1
1.1 INTRODUCTION 1
1.2 RESEARCH OVERVIEW 1
1.2.1 WORKFLOWS 1
1.2.2 TYPES OF WORKFLOW AND WORKFLOW ENGINES 2
1.2.3 WORKFLOW MODELLING TECHNIQUES AND ARCHITECTURE USED IN WORKFLOW ENGINES 3
1.2.4 THE BLACKBOARD PARADIGM 3
1.2.5 UTILIZING THE BLACKBOARD PARADIGM TO BUILD A WORKFLOW ENGINE 4
1.3 ORGANISATION OF THE REMAINDER OF THE DISSERTATION 5
1.4 SUMMARY 6
2 BACKGROUND TO WORKFLOW MANAGEMENT SYSTEMS 7
2.1 INTRODUCTION 7
2.2 WORKFLOW MODEL 7
2.3 WORKFLOW MODELLING TECHNIQUES 8
2.4 WORKFLOW PATTERNS 10
2.5 DATA PATTERNS 14
2.6 WORKFLOW MANAGEMENT SYSTEMS AND WORKFLOW ENGINES 14
2.7 WORKFLOW PROCESS COMPONENTS 15
2.8 WORKFLOW MODELLING TECHNIQUES AND ENGINE IMPLEMENTATIONS 18
2.8.1 WORKFLOW MODELLING AND LANGUAGES 18
2.8.2 GRAPH-BASED WORKFLOW MANAGEMENT SYSTEMS 19
2.8.3 BPEL AND WORKFLOW ENGINE IMPLEMENTATIONS 20
2.8.4 PETRI-NET BASED WORKFLOW ENGINE IMPLEMENTATIONS 22
2.8.5 SCRIPT BASED WORK LANGUAGES AND WORKFLOW ENGINE IMPLEMENTATIONS 22
2.8.6 GENERAL ANALYSIS AND OBSERVATIONS 24
2.9 SUMMARY 25
3 USING THE BLACKBOARD PARADIGM IN THE CONTEXT OF A WORKFLOW ENGINE 26
3.1 INTRODUCTION 26
3.2 BLACKBOARD PARADIGM 26
3.2.1 SPECIALISTS 27
3.2.2 THE BLACKBOARD 27
3.2.3 NEED FOR CONTROL 28

3.2.4 Event-Based Blackboard Systems 30
3.2.5 External Data Insertion on the Blackboard 31
3.2.6 Serial vs. Concurrent Blackboard Systems 31
3.2.7 Blackboard Design Stipulations or Guidelines 34
3.3 Petri-Net Conceptualization of a Workflow Process 35
3.4 Mapping a Petri-net Modelling Approach to the Blackboard Paradigm 36
3.5 Serial Vs Concurrent Blackboard Implementation 39
3.6 Prior Research Concerning Workflow Engines and the Blackboard Paradigm 41
3.7 Summary 42
4 Research Question 43
4.1 Introduction 43
4.2 Research Scope 43
4.3 Aspects of Workflow Management and Quality Indicators 44
4.3.1 Behavioural Aspect 44
4.3.2 Informational Aspect 45
4.3.3 Other Aspects 46
4.4 Research Question 48
4.5 Research Contributions, Methodology and Validation 48
4.6 Summary 49
5 The Prototype - Wits Enterprise Workflow Engine (WEWE) 50
5.1 Introduction 50
5.2 The Blackboard 50
5.2.1 Overall Structure 50
5.2.2 Implementation 52
5.3 The Specialists 55
5.3.1 Rule Based Systems 55
5.3.2 Specialist Rules Implementation 56
5.3.3 Inactive and Active Specialists 60
5.3.4 Specialist Creation and Persistence 61
5.3.5 Specialist Actions Implementation 62
5.4 Blackboard Agents 63
5.4.1 Agent Implementation 64
5.4.2 Types of Agents Created 65
5.4.3 Comparison with Specialists 67
5.4.4 Agents and Interoperability of WFMSs 67
5.5 Orchestrator/Controller 68
5.5.1 WF Process Management 68
5.5.2 WF Analytics and Management 69
5.5.3 Concurrent Specialist Triggering 69

5.5.4 Specialist Timeout Triggering 71
5.5.5 Agent Execution Triggering 71
5.5.6 Sub-Workflow and Composite Workflows 72
5.5.7 Ad-hoc Workflows 73
5.6 Workflow Designer 74
5.7 Interaction of WF Engine Components 74
5.8 Summary 78
6 Analysis and Testing 79
6.1 Introduction 79
6.2 WF Patterns 79
6.2.1 Basic Control Flow Patterns (WF Patterns 1-9) 80
6.2.2 Structural Patterns (WF Patterns 10-15) 83
6.2.3 State Based Patterns (WF Patterns 16-18) 87
6.2.4 Cancellation Patterns (WF Patterns 19-20) 91
6.2.5 Final Results and Behavioural Comparison with Existing Workflow Engines 91
6.2.6 Implementation Comparison with Existing Workflow Engines 92
6.3 Data Patterns 93
6.3.1 Data Visibility Patterns (1-8) 93
6.3.2 Internal Data Interaction Patterns (9-14) 95
6.3.3 External Data Interaction Patterns (15-26) 96
6.3.4 Data Transfer Patterns (27-33) 98
6.3.5 Data Routing Patterns (34-40) 100
6.3.6 Final Results and Comparisons with Existing Workflow Engines 102
6.4 Summary 102
7 Conclusion 104
7.1 Introduction 104
7.2 Summary of Work Presented 104
7.3 Findings 106
7.3.1 Support for the Behavioural Aspect of WF Engines 106
7.3.2 Support for the Informational Aspect of WF Engines 106
7.3.3 Final Finding 107
7.4 Future Work 107
8 References 109
9 Bibliography 115
Appendices 118
A. Blackboard Implementation 118
A.1 Introduction 118
A.2 Blackboard Structure 118
A.3 Blackboard Database Schema 120

A.4 DATA TYPES 123
A.5 CONCLUSION 125
A.6 REFERENCES 125
B. SPECIALIST IMPLEMENTATION 126
B.1 INTRODUCTION 126
B.2 SPECIALIST CREATION AND PERSISTENCE IMPLEMENTATION 126
B.3 SPECIALIST ACTION EXECUTOR IMPLEMENTATION 133
B.4 CONCLUSION 134
B.5 REFERENCES 134
C. WF DESIGNER 135
C.1 INTRODUCTION 135
C.2 CREATING A WORKFLOW USING WEWE'S WF DESIGNER 135
C.3 CONCLUSION 147
D. WF EXAMPLE TO DEMONSTRATE WF ORCHESTRATION 148
D.1 INTRODUCTION 148
D.2 DESIGNING THE SPECIALISTS 148
D.3 WF PROCCESS ORCHESTRATION 149
D.3.1. WF SPAWN 149
D.3.2. ACCESS CLAIM ACTIVITY 150
D.3.3. PROCESS CLAIM ACTIVITY 152
D.3.4. NOTIFY CLIENT ACTIVITY 154
D.4 CONCLUSION 156

List of Tables

Table 1: Workflow patterns 1-9 and their support rating. 80
Table 2: Workflow patterns 10-15 and their support rating. 84
Table 3: Workflow patterns 16-18 and their support rating. 87
Table 4: Workflow patterns 19-20 and their support rating. 91
Table 5: Data patterns 1-8 and their support rating. 94
Table 6: Data patterns 9-14 and their support rating. 95
Table 7: Data patterns 15-26 and their support rating. 96
Table 8: Data patterns 27-33 and their support rating. 99
Table 9: Data patterns 34-40 and their support rating. 100
Table A.1: The schema for the table wewe_workflows. 120
Table A.2: The schema for the table wewe_process_instances. 121
Table A.3: A list of all workflow process instance statuses and their descriptions. 121
Table A.4: The schema for the table wewe_wf_process_constants 122
Table A.5: The schema for the table wewe_wf_process_iterations 122
Table A.6: The schema for the table wewe_wf_process_variables. 122
Table A.7: Data types supported by WEWE. 124
Table B.1: The schema for the table wewe_specialists. 126

List of Figures

Figure 1: A simple view of the blackboard paradigm. 4
Figure 2: An example of a sequential Petri-net [21]. 9
Figure 3: A hypothetical workflow. (a) A normal flowchart of a hypothetical workflow. (b) A workflow graph of a hypothetical workflow. 10
Figure 4: The Sequence workflow pattern. 11
Figure 5: The Parallel Split workflow pattern. 11
Figure 6: The Synchronization workflow pattern. 11
Figure 7: The Exclusive Choice workflow pattern. 12
Figure 8: The Multi-Merge workflow pattern. 12
Figure 9: The Differed Choice workflow pattern. 13
Figure 10: The Interleaved Paralleled Routing workflow pattern. 13
Figure 11: The WFMS reference model of the Workflow Management Coalition [27]. 15
Figure 12: A workflow flowchart for an employee requesting vacation leave. 16
Figure 13: Deployment process for most BPEL workflow engines. 21
Figure 14: The complete view of the blackboard paradigm adapted from [65] 29
Figure 15: The concurrent blackboard implementation approaches. (a) Distributed blackboard approach. (b) Blackboard server approach. (c) Shared memory blackboard approach. 33
Figure 16: A Petri-net illustrating the Parallel Split workflow pattern. 36
Figure 17: The relationship between process rules, process data and process actions to be executed. 36
Figure 18: The relationship between process rules, process data and process actions to be executed in a blackboard paradigm. 37
Figure 19: A comparison between the linear flow of a traditional workflow engine and the cyclic flow of an engine built on the blackboard paradigm. 38
Figure 20: Parallel workflow process example. 40
Figure 21: The architecture designed in [83] employs elements of layered as well as blackboard architecture. 41
Figure 22 : A UML diagram of the fundamental relationship between workflows and process instances. 51
Figure 23: The conceptual structure of WEWE's blackboard. 52
Figure 24: UML diagram of the blackboard structure. 53
Figure 25: Structure of a workflow process instance. 54
Figure 26: Components of a Rule Based System. 56

Figure 27: The process undertaken by specialists when they become active. 61
Figure 28: A typical scenario of a user interacting with WEWE's workflow client application. 64
Figure 29: Agent transport process between two components within a WFMS. 65
Figure 30: Producer-consumer pattern was used for specialist triggering. 70
Figure 31: The way in which the mediator pattern is used to handle agents updating the blackboard. 72
Figure 32: Hypothetical workflow of approving or rejecting an insurance claim. 75
Figure 33: Sequence diagram for the workflow involved in approving and processing an insurance claim. 76
Figure 34: The AND Split WF pattern. 81
Figure 35: Example of looping using the blackboard paradigm. 85
Figure 36: Example of WF pattern 16. 89
Figure 37: Example of WF pattern 17. (a) Direct implementation of WF pattern 17. (b) Indirect implementation of WF pattern 17. 89
Figure 38: Example of WF pattern 18. (a) Direct implementation of WF pattern 18. (b) Indirect implementation of WF pattern 18. 90
Figure A.1: The Entity Relationship Diagram (ERD) of the fundamental relationship between workflows and process instances. 118
Figure A.2: The structure of WEWE's blackboard with implementation details. 119
Figure A.3: UML diagram of the blackboard structure with implementation details. 120
Figure A.4: The relationship between WF process instances and the data type manager. 124
Figure B.1: The process undertaken by the specialist animator whenever an action is executed. 132
Figure C.1: Login Screen. 135
Figure C.2: Home page. 136
Figure C.3: How to start designing a new WF. 136
Figure C.4: Screen for inserting WF metadata. 137
Figure C.5: Screen for setting WF dependencies and constraints. 137
Figure C.6: As an example a screen to insert a new constant WF constraint. 138
Figure C.7: As an example a screen to new constant being defined as a WF constraint. 138
Figure C.8: The screen to insert interfaces for the WF. 139
Figure C.9: The screen to create specialists for the WF. 139
Figure C.10: The screen to insert metadata for the new specialist. 140
Figure C.11: The specialist main menu. 140
Figure C.12: The specialist attribute main menu. 141
Figure C.13: The specialist attribute creation menu. 141
Figure C.14: The specialist rule creation screen. 142
Figure C.15: The specialist rule antecedent modal window screen. 142
Figure C.16: Antecedent drag and drop screen. 143
Figure C.17: Setting antecedent conjunctions. 143

Figure C.18: An example of selecting the 'AND' keyword. 144
Figure C.19: The actions creation screen. 144
Figure C.20: The action creator modal window. 145
Figure C.21: The action parameters setter modal window. 145
Figure C.22: The drag and drop action function. 146
Figure C.23: The specialist attributes menu when attributes have been added. 146
Figure C.24: The specialist screen with one specialist created. 147
Figure D.1: Hypothetical workflow of approving or rejecting an insurance claim. 148
Figure D.2: Initial blackboard data when the process is spawned. 149
Figure D.3: Specialist A interacting with the blackboard. 150
Figure D.4: Agents interacting with the blackboard to trigger an action. 151
Figure D.5: Specialist A interacting with the blackboard for the second time. 152
Figure D.6: Specialist B interacting with the blackboard. 153
Figure D.7: An agent interacting with the blackboard to trigger the notify client action. 154
Figure D.8: Specialist C interacting with the blackboard to execute the last WF activity in the WF. 155

List of Abbreviations, Acronyms and Symbols

Abbreviations, Acronyms and Symbols	**Meaning**
WfMC	Workflow Management Coalition
WEWE	Wits Enterprise Workflow Engine
REST	REpresentational State Transfer
SOAP	Simple Object Access Protocol
BPEL	Business Process Execution Language
WFMS	Workflow Management System
YAWL	Yet Another Workflow Language
GUI	Graphical User Interface
JWT	Java Workflow Tooling
WF	Workflow
CRUD	Create, Read, Update and Delete
FIFO	First In First Out
WAPI	Workflow Application Programming Interface
GUID	Global Unique Identifier
DBMS	Database Management System
WS-BPEL	Business Process Execution Language for Web Services
XML	Extensible Markup Language
POC	Proof of Concept
SOA	Service Oriented Architecture
PHP	Hypertext Preprocessor
UI	User Interface
OSS	Open Source Software
PVM	Process Virtual Machine
BPM	Business Process Management
JSON	JavaScript Object Notation
RBS	Rule Based System
LDAP	Lightweight Directory Access Protocol

1 Introduction

1.1 Introduction

Workflow Management Systems are used for automating and controlling business processes [1]. A traditional definition for a workflow is provided by the Workflow Management Coalition [2] which states: “Workflow is the computerized facilitation or automation of a business process, in whole or part”. Workflow systems implement the principles of automatic control in business systems. Automatic control does not necessarily encompass solely software or technical components but rather includes the integration of both human and software activities and manual interactions. Workflow management has evolved into a mature field with numerous workflow management systems with scores of features. New and upgraded workflow managements systems are continually being released to automate business processes of users and organizations that are becoming increasingly complex. It becomes difficult for users to choose the pertinent workflow engine for their required business processes [3]. Moreover, it becomes difficult to customize and to extend these workflow engines which make integration with existing applications difficult [3]. This usually results in custom in-house solutions being developed. The architecture and operational benefits of the blackboard paradigm can tentatively be seen to be compatible with the principles on which workflows engines are based. Using solely the blackboard paradigm, an extensible and customizable workflow engine might be developed to satisfy the needs and requirements of most users. The aim of this research is to investigate this very supposition.

1.2 Research Overview

1.2.1 Workflows

The concept of workflows has evolved from a means to describe the flow of paperwork through an organization to a more abstract and general technique used in many application domains such as business processes, scientific applications, and e-learning. Currently, workflows are most often considered a programmatic structure for designing and executing processes [4].

Workflows are construed in such a way as to enable a clear separation between business logic and its implementation. The business logic can be defined in a workflow language while the actual implementation is done in a traditional programming language. A plethora of workflow languages have been developed to define these workflow processes and patterns. Two examples of these are Business Process Execution Language [5] (BPEL) and Yet Another Workflow Language [6] (YAWL). Basically, the 'what' is done in a workflow and the 'how' is done in traditional programming code.

To create a workflow, a *process* has to be defined that later can be translated into a workflow which can be understood by a workflow engine [7].

1.2.2 Types of Workflow and Workflow Engines

Workflows can be characterized as being administrative, ad hoc or production workflows depending on differing degrees of structure, repeatability and control. Administrative workflows are highly repetitive and consist of an established set of rules or process definitions such as an application form being authorized by several people. In most cases, a form has to be filled out to begin this type of workflow [8].

Ad hoc workflows can be seen as an aberration to administrative workflows in which the rules or process definitions are not rigidly defined in advance. An ad hoc workflow can be created if an administrative workflow was altered for a specific case or a unique exception. For instance, a workflow can be created for reviewing papers that typically requires three reviewers. However, a special case can arise that requires a review process with five reviewers instead of three. Workflow systems that offer the creation and running of ad hoc workflows provide flexibility to seamlessly alter general workflow process templates when unique circumstances arise [9].

Production workflows are used to implement critical business workflow processes such as an approval of an insurance claim or a loan application. This type of workflow differs from administrative workflows only in complexity. Administrative workflows depend mainly on humans to make decisions and perform actions while production workflows are able to retrieve relevant information across a heterogeneous software environment to make decisions and perform actions [3].

A comprehensive workflow management system should be able to support these three main workflow types.

With workflow engines being developed to satisfy the needs of their users, two distinct types of workflow (WF) engines have been created which are scientific [10] and business [11] workflow engines. Scientific workflow engines are developed for automating scientific experiments that need to deal with large amounts of data. Business workflow engines are developed to run processes within a business environment that have to deal with events within a workflow process.

1.2.3 Workflow Modelling Techniques and Architecture Used in Workflow Engines

There are many kinds of techniques to model workflows. Two examples of these are Petri-nets [12] and graph based modelling [13]. The concepts behind these modelling techniques help lay the foundation for how workflow engines are built. Many workflow engines can be categorized based on the types of modelling techniques used. However, there are also empirically derived approaches to model workflows such as using workflow [14] and data [15] patterns. These patterns express recurring scenarios that are found in workflows and can be used in building workflow engines and designing workflows. They can also be used to benchmark any workflow engine by determining how many workflow and data patterns can be supported by that particular workflow engine. Existing research has been conducted that have critiqued many commercially available workflow engines. It is found that many of these workflow engines struggle to support complex workflow and data patterns.

Workflow engines are complex software applications and therefore most workflow engines scrutinized in existing research do not have single pure architectures but rather consist of a heterogeneous architecture. Furthermore, none of these commercially available workflow engines take the blackboard approach.

1.2.4 The Blackboard Paradigm

The blackboard paradigm is used as a flexible problem solving approach and it utilizes the opportunistic problem solving model. Blackboard architectures support multiple problem solving techniques, multilevel abstraction of situations and control processes and responsive opportunistic control of workflow process activity [16].

The blackboard architecture is popularly described as a set of people in front of a large blackboard. These people or specialists have to work together and produce a solution to a problem, using the blackboard as the workplace for cooperatively developing the solution. The process begins with a problem specification written onto a blackboard. The specialists all watch the blackboard, looking for an opportunity to apply their expertise to the developing solution. When one specialist writes something on the blackboard, this allows another specialist to apply their expertise. The second specialist then records their contribution on the blackboard, hopefully enabling other specialists to then apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved.

Figure 1 depicts a simple view of the blackboard architecture. From Figure 1, it can be seen that the blackboard paradigm consists of two kinds of components namely a central data structure that embodies the current state (blackboard) and independent components (specialists) that operate on the blackboard. The specialists are independent components of knowledge. It is important to note that the interactions between the specialists only take place through the blackboard. Specialist do not interact which each other directly. The blackboard contains the problem solving data and specialists can alter the blackboard to incrementally arrive to a solution to the problem. It can be seen that there exists no central control component. Control is intrinsically driven by the state of the blackboard. Specialists apply their knowledge opportunistically if the state of the blackboard allows them to [16].

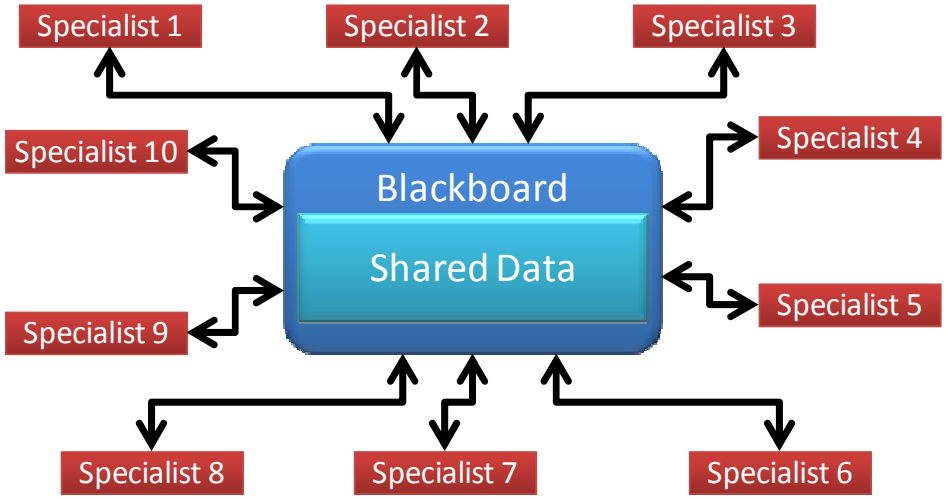

Figure 1: A simple view of the blackboard paradigm.

1.2.5 Utilizing the Blackboard Paradigm to build a Workflow engine

The opportunistic control innate to blackboard systems can possibly be leveraged in order to support complex workflow and data patterns. Additionally, the architecture and operational benefits of the blackboard paradigm can be seen to complement the inner precepts of how workflow engines work. This research aims to verify whether or not the blackboard paradigm can solely be used to build a workflow engine.

Research has already been conducted on the feasibility of building a workflow engine using the blackboard paradigm [16]. The systems presented in the existing research do not use the pure blackboard paradigm and have incorporated some aspects of layered architecture as well as retaining some aspects of blackboard architecture. They negate opportunistic control in favour of a more rigid approach in controlling and executing workflows. In contrast, this research is about being true to the blackboard paradigm.

To verify this research, a prototype workflow engine is developed solely utilizing the blackboard paradigm. The main decisive factor for research verification is to determine the level of support of workflow and data patterns the prototype can provide. Test cases are created and executed within the prototype in order to verify the level of support for workflow and data patterns. Besides this, the prototype is also tested for the presence of other quality indicators provided by prominent researchers within the field of workflow management. The next section provides a guide to the remainder of the dissertation.

1.3 Organisation of the Remainder of the Dissertation

The remainder of the dissertation is presented as follows:

Chapter 2: Some background information surrounding the relevant topics is presented. The main components and general behaviour of workflows is discussed and some techniques to model workflows which appear in the literature are listed. An empirical approach to model workflows using workflow and data patterns is mentioned. The distinctions between workflow management systems and workflow engines are made. Many different kinds of leading workflow engines are examined and categorized based on the kinds of workflow modelling techniques they use.

Chapter 3: This chapter shows how the blackboard paradigm can be used to build a workflow engine. The inner components and workings of the blackboard paradigm are unpacked. A conceptual model of a workflow is formulated and this conceptual model is mapped to the different components of the blackboard paradigm. Serial and concurrent implementations of the blackboard paradigm are compared. An implementation comparison between a workflow engine built using a blackboard paradigm and other contemporary workflows engines present in the market is provided.

Chapter 4: The research question and contributions are presented. A broad scope is present in the arena of workflow management and it becomes difficult to define a set of criteria to evaluate a workflow engine built using the blackboard paradigm. Researchers who are prominent in this field have already addressed this issue and have provided five main aspects of workflow management and their implementation requirements and quality indicators in order to evaluate any workflow engine. The five main aspects are discussed in detail and they are the functional, behavioural, informational, operational and organizational aspects of workflow management. These main aspects of workflow management are used to formulate a research question. The research objectives and methodology are laid out which broadly entails the building of a prototype workflow engine utilizing the blackboard paradigm and verifying the presence of the quality indicators put forth by the main aspects of workflow management within the prototype.

Chapter 5: The prototype that was designed and developed to validate this research is presented. The design and implementation details of each component within the blackboard paradigm are unpacked. Software agents which are components not integral to the blackboard paradigm but are required within the workflow engine are introduced. The WF designer which allows users to create workflows is discussed and a complete picture of the blackboard paradigm is presented by using a real world workflow example to show how the blackboard system operates in order to execute workflows.

Chapter 6: The test results and analysis of the prototype is presented in Chapter 6. The results of workflow and data pattern tests are presented and analyzed. Additionally the workflow and data pattern results from this prototype

are compared with existing workflow engines currently on the market. Positive aspects and some limitations of the workflow engine prototype are presented.

Chapter 7: This chapter concludes this dissertation by reiterating the salient points made. The research question is restated and all the research findings are summarized in order the answer the research question. Lastly, possible future work and avenues that future research can take are listed.

Additional information about the prototype can be found in the appendices. These appendices are organized as follows:

Appendix A: The actual implementation and database schema for the blackboard component of the workflow engine is explained.

Appendix B: Additional implementation details of the specialist component within the workflow engine are explained.

Appendix C: A user manual with screenshots of the WF designer that was developed for the prototype is presented.

Appendix D: A real world workflow example is used to show how the prototype functions on a step-by-step basis.

1.4 Summary

This chapter introduces the research and presents how the remainder of this dissertation is organized on a chapter-by-chapter basis.

The next chapter provides information from relevant literature in order to contextualize this research.

2 Background to Workflow Management Systems

2.1 Introduction

Chapter 1 has introduced the research and has provided a general structure of the entire dissertation. This chapter presents background information from existing literature in the pertinent areas of research. The main components and behaviour of workflows is given. Some workflow modelling techniques are listed. An explanation on how workflows are executed inside workflow engines is provided. The internal architecture, workflow modelling techniques used and implementation details of some leading workflow engines are unpacked.

2.2 Workflow Model

A process has been defined by the Workflow Management Coalition (WfMC) as a set of tasks or actions, the order in which they must be performed, permissions defining who can perform them, and a script that is executed for each action. Additionally, the process must consist of *process data* to define start and end conditions and flow control logic [17].

The purpose of a workflow is to enable the execution of *tasks* or *process actions* in a specified order. Process actions are handed over to a *resource* to be executed which is either a machine or a human. It is not always necessary for a process action to be executed immediately if the workflow has enabled its execution. The execution of a process action is sometimes dependant on a human to be completed or is executed when a timeout occurs. For instance, if a workflow process entails the sending of a form to an employee to be processed, the subsequent corresponding actions cannot be executed until the employee submits the form. Additionally, a timeout pre-condition could be attached to this action which states that if the employee has not submitted this form within a predefined time, escalate this issue to the employee's supervisor.

This subtle differentiation between the enabling and the executing of a task revolves around the concept of *triggering* [18]. A trigger is a condition which dictates when a task is to be executed. Typically, there are four types of triggers:

- Automatic: A task is executed immediately as soon as it is enabled. Human interaction is not required.
- User: A task is triggered by a human user. The task might be enabled beforehand so that the human participant can actually execute it.
- Message: An external event or message triggers a previously enabled task. A message could be in the form of an email or fax message, REST or SOAP call or an HTTP request.
- Time: A task is triggered by a timer. The timer can be set to timeout at some predefined time.

Once a particular workflow process action has been executed, it might generate some data that might be required in subsequent actions within the business process to be executed successfully. This type of data is termed as *production data* (data used and produced by the business process). It is distinct from *control data* that is produced and used by the workflow engine to execute a workflow process instance. Both of these different types of data are stored inside the *workflow process data*.

A *workflow process definition* stipulates the flow of control between tasks and the order of tasks being executed. The routing between tasks can be defined by *process rules* which capture aspects of workflow process control. Wil van der Aalst in his influential paper entitled 'Workflow Patterns' has identified 20 main workflow patterns which encapsulate the various ways in which tasks can be routed and how workflow processes are controlled [14]. Much like software patterns, workflow patterns express recurring scenarios that are found in workflows. Workflow Patterns will be unpacked in more detail later (see Section 2.4). To recap, a workflow process can be decomposed into workflow process rules, process data and process actions.

2.3 Workflow Modelling Techniques

There are several modelling techniques used to model workflows. A few are listed below:

- State charts which are derived from Finite State Machines (FSMs). This modelling technique uses states and transitions determined by Event Condition Action rules (ECA rules) to define the control flow between activities [19].
- UML activity diagrams which are derived from state charts. This modelling technique specifically defines the control flow from one activity to the next without the need to formally define states [20].
- Petri-nets which is a formal mathematical modelling language ideal for representing concurrent behaviour in discrete distributed systems [21].

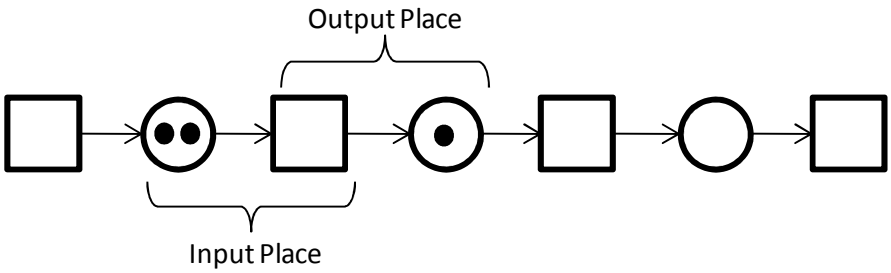

Figure 2: An example of a sequential Petri-net [21].

Van der Aalst has proposed that Petri-nets should be used for modelling workflows. The rationale behind this proposal is that "Petri-nets have formal semantics despite their graphical nature and an abundance of analysis techniques exist" [21]. Petri-nets consist of *places* (circles), *transitions* (squares), *arcs* (arrows) and *tokens* (dots). Only two kinds of Petri-nets can be constructed that is when a place is connected to a transition via an arc or vice versa. These two fundamental units are called an input place and an output place respectively. Figure 2 shows an example of a sequential Petri-net. From Figure 2, it can be seen that places can consist of many tokens to signify the state of the Petri-net which is termed as the *marking*. A Petri-net can be activated by the *firing* of transitions. Firing of a transition entails a transition consuming tokens from its input place, performing a set of actions and placing the tokens in its output places. This firing of a transition (three actions) happens atomically. Any transition within a Petri-net may fire at any time provided there are tokens within the input places (non-deterministic behaviour). This makes Petri-nets ideal for modelling concurrent behaviour in discrete distributed systems.

Graph based workflow modelling is another approach to model workflows. A graph is a data structure where a set of nodes are connected together by lines or edges. In the context of workflows, the nodes can be seen as workflow activities and the lines can be seen as the transitions between workflow activities. Workflows can be represented as Directed Acyclic Graphs (DAGs). A DAG is a kind of graph where nodes and lines are connected together in such a way that any path traversed using the lines emerging from a particular node will never reach that node again [22]. A flowchart of a hypothetical workflow such as the one seen in Figure 3(a) can actually be construed as a DAG.

From Figure 3(a), workflow actions A and B are executed successively and thereafter workflow actions C and D are executed concurrently followed by action E being executed thereafter terminating the workflow. However, it is not clear when workflow action E should be executed. There are three possibilities. One is action E should be executed when both workflow actions C and D have been executed. The next possibility is whenever any one of the two actions (workflow actions C or D) have been executed, action E should be executed. The last possibility could be that action E should be executed only once when either one of the two workflow actions C and D have been executed. These three types of workflow execution cases are called the synchronization, simple merge and discriminator constructs respectively. Similarly, it is also unclear when and in what fashion workflow actions C and D should be executed after the workflow action B has been executed. This has lead to the development of workflow graphs that incorporate constructs specific to workflows into DAGs to help model workflows adequately [23].

Figure 3(b) shows a workflow graph that models the same hypothetical workflow. It can be seen that special workflow constructions such as the fork and synchronization constructs exist to clearly define the path a workflow process can take.

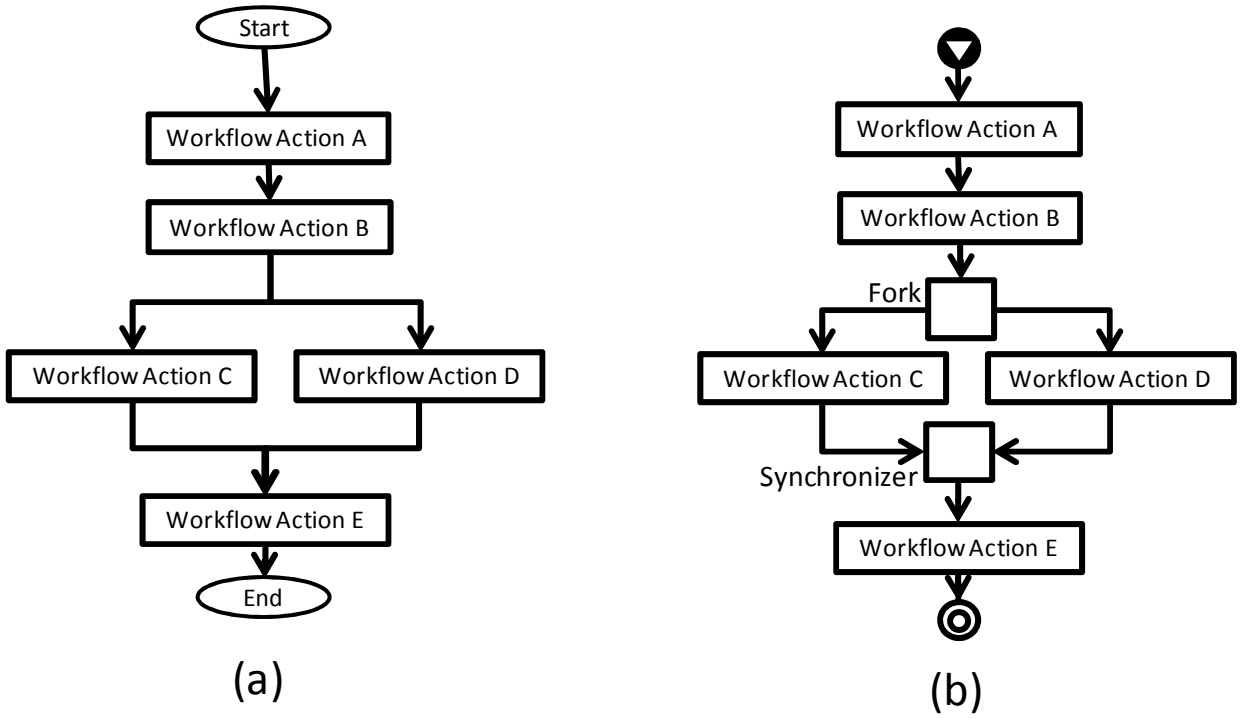

Figure 3: A hypothetical workflow. **(a)** A normal flowchart of a hypothetical workflow. **(b)** A workflow graph of a hypothetical workflow.

Petri-nets and workflow graphs along with any other modelling techniques provide more abstract approaches for modelling different kinds of workflows. Despite there being more pragmatic approaches to define workflows which will be discussed later (see Section 2.4), it is nevertheless important to understand how these modelling techniques orchestrate workflows and how activities inside a workflow are executed based on some sort of control flow. The concepts behind them help to lay the foundation for how to build workflow engines (see Section 2.8).

2.4 Workflow Patterns

Workflow patterns express recurring scenarios that are found in workflows and are applicable for both building and use within workflow engines. They can be used to benchmark any workflow engine by determining how many workflow patterns can be run by that particular workflow engine. The user of a workflow engine can also use workflow patterns as a reference point to build workflows. A few of these workflow patterns are listed below as examples. They are documented in [14]:

- *Sequential Routing:* One activity in the workflow process will be enabled after the completion of another activity in the same process. Figure 4 shows an example of this where activity B is executed after activity A

has been completed. The squares in the figure signify the workflow activities and the arrows signify the transition between two workflow activities.

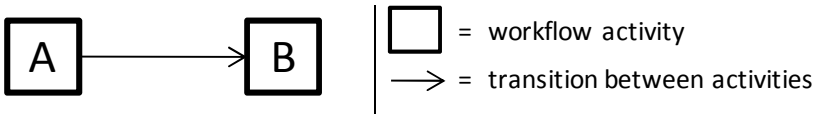

Figure 4: The Sequence workflow pattern.

- *Parallel Split:* A point in the workflow process in which a single thread is split into multiple threads which can be executed in parallel. From Figure 5, activities B, C and D are executed all at once after activity A has been completed.

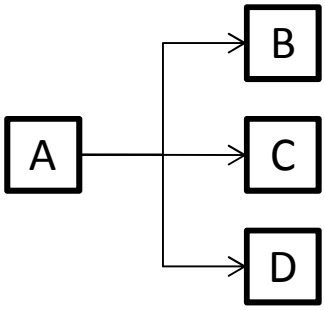

Figure 5: The Parallel Split workflow pattern.

- *Synchronization:* A point in the workflow process where multiple parallel threads converge once all have been completed into a single thread of execution. Figure 6 show an example where action D is executed once after the concurrent activities A, B and C are completed.

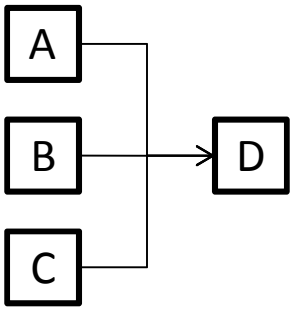

Figure 6: The Synchronization workflow pattern.

- *Exclusive Choice:* A point in the workflow process where based on a decision or workflow flow data, one or several threads are chosen to be executed. Figure 7 shows an example where only action D is chosen to be executed after action A has been executed.

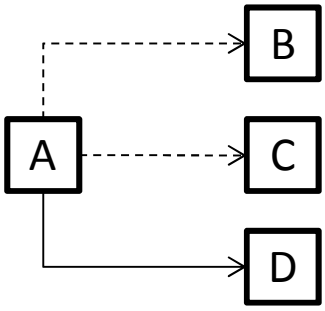

Figure 7: The Exclusive Choice workflow pattern.

- *Multi-Merge:* A point in the workflow process where multiple parallel threads converge without synchronization. If multiple threads are running concurrently, the workflow activity after the merge is executed for every incoming thread. Figure 8 shows an example where action D is executed for every incoming thread.

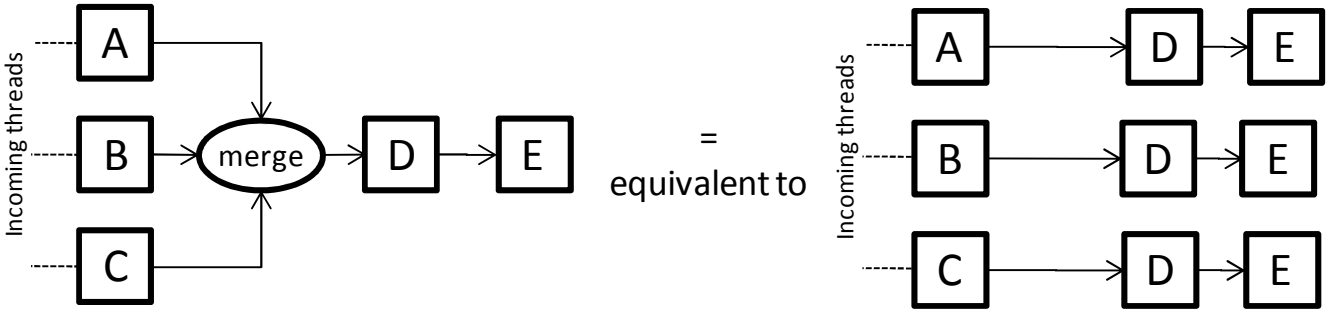

Figure 8: The Multi-Merge workflow pattern.

The examples provided above are classified as basic flow patterns. They are rigidly defined during design time. Many other workflow patterns are more complex and are flexible enough to be altered by external stimuli during runtime. These kinds of workflow patterns are heavily dependent on the state of the workflow. A few examples of these workflow patterns are given below:

- *Differed Choice:* A point in the workflow process where one of many enabled branches is selected. There is race between the many enabled branches and only one branch can be executed. Once the workflow environment selects to execute one of the enabled branches through any number of means (human interaction, external message, timeout, etc), the other enabled branches will be withdrawn or cancelled. This selection occurs during runtime. Figure 9 shows two branches with workflow activities B and C being enabled. The selection of executing any one workflow activity first will withdraw or cancel the other activity.

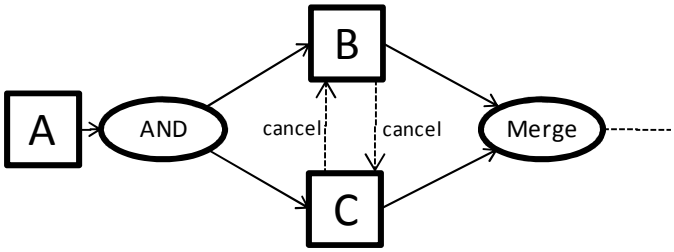

Figure 9: The Differed Choice workflow pattern.

- *Interleaved Parallel Routing:* A point in the workflow where a set of activities can be executed in any order. The order of execution is decided at runtime and only one activity can be executed at any one time. Figure 10 shows that workflow activities B, C and D are enabled to be executed in any order. Once the entire set of enabled activities is executed once in any order, the workflow will continue.

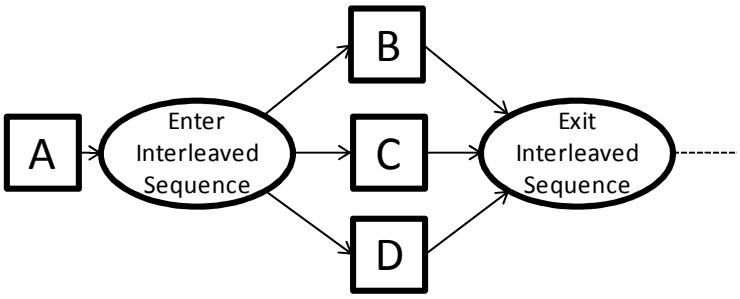

Figure 10: The Interleaved Paralleled Routing workflow pattern.

- *Cancel Activity:* A point where an enabled workflow activity is disabled.

- *Cancel Case:* A point where an instance of a running workflow process is cancelled.

These workflow patterns determine the level of workflow functionality and provide a way to compare workflow engines. Van der Aalst in [14] did a comparison of the level of support for the 20 main workflow patterns as provided by 15 leading workflow engines in 2003. At that stage these workflow engines could only implement 51.6% of the 20 main workflow patterns. However, the unsupported workflow patterns could be realized by implementing a combination of other workflow patterns but this increases the workflow designer's implementation effort. The same investigation was conducted by Van der Aalst and ter Hofstede in 2006 against later versions of the same workflow engines and new workflow engines released into market namely Oracle BPEL, SAP Workflow and XPC [24]. The average number of workflow patterns that were implemented was raised to 58.4%. The same authors of [14] and [24] have provided a website called the "workflow patterns initiative" in which they have continued their study [25]. They have applied their workflows pattern tests to new versions of the same workflow engines and some new ones (17 workflow engines in total) as of 2013 and found that 64.1% of the workflow

patterns could be implemented. A gradual improvement in support can be seen. However, it is noteworthy that no workflow engine in this collection can support all the 20 main workflow patterns.

Besides the 20 main workflow patterns mentioned in [14], a further 22 more workflow patterns are also defined in [24] that are essentially extensions and special cases to the main 20 workflow patterns.

2.5 Data Patterns

Data patterns have been introduced by Russell, ter Hofstede and Edmond to the workflow management arena in order to evaluate the extent to which workflow engines can represent, transport and utilize data within a workflow [15]. Just as workflow patterns capture types of control flow, data patterns capture types of data flow. Forty main data patterns have been identified and they were used to evaluate six leading WFMSs in 2005. At that stage it was found that 48.3% of the data patterns could be implemented. The same authors have continued this study on the "workflow patterns initiative" website and have applied their data patterns to new versions of the same workflow engines and many new ones (17 workflow management systems in total) and found that 51.2% of the data patterns can be implemented [25]. An improvement in support over time can be seen.

2.6 Workflow Management Systems and Workflow Engines

A workflow is a set of tasks or actions, the order in which they must be performed, permissions defining who can perform them, and a script that is executed for each action. A Workflow Management System (WFMS) supports the execution and control of these workflow processes. It allows users, more specifically process designers to create processes and rules to govern these processes either through a Graphical User Interface (GUI) or a high level workflow language. A single workflow process will be a series of activities and the control structures to connect them. If a graphical tool such as a workflow designer exists within the workflow engine, the process designer will be able to draw flowcharts to define a process and these flowcharts will determine how the workflow behaves [26].

It is tempting to conflate the terms workflow engine and Workflow Management System. However, these are two distinct entities. A workflow engine's main purpose is to execute and control workflow processes. The workflow management system consists of many components including the workflow engine. The Workflow Management Coalition (WfMC) has provided a reference model of a WFMS shown in Figure 11 [27]. It can be seen that the core component of the WFMS is the workflow engine which is accessed through a workflow API that is available to resources and applications. The workflow engine executes actions within workflow processes by invoking the interfaces provided by external applications. These interfaces are housed in the workflow API (WAPI) so they can be readily accessible to the workflow engine. The process definition tools are utilized to design new workflow process definitions generally through a graphical user interface. Workflow Management Systems might also provide a simulator to analyze workflow process definitions. The administration and monitoring tools are available

to monitor running workflows. The end-user interacts with the WFMS through workflow client applications. Popular examples of workflow client applications are workflow list and in-basket [28]. The architecture of many WFMSs roughly follows this reference model shown in Figure 11 [27].

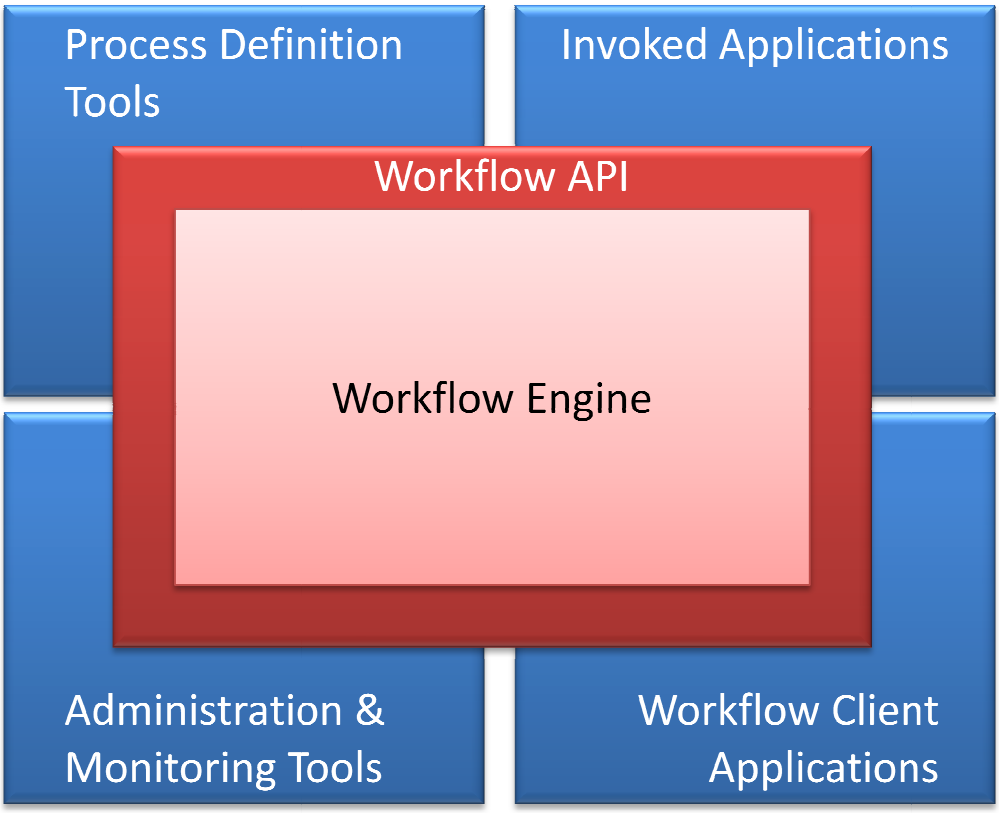

Figure 11: The WFMS reference model of the Workflow Management Coalition [27].

2.7 Workflow Process Components

While a workflow process is running inside a workflow engine, a sequence of actions or activities are executed. The workflow engine has the capability to transition from one activity to the next. These transitions can occur sequentially or a decision node or rule can be placed between them that can determine whether the next activity should be allowed to be executed. All of these transitions within the workflow can be captured by any one of the many workflow patterns.

For instance, a workflow can be created inside a workflow management system that enables employees to apply for vacation leave. A flowchart of this workflow is shown in Figure 12. Once the employee submits a form with details of the leave, an instance of the workflow, namely a workflow process, starts inside the workflow engine. The workflow engine will send the details of the vacation leave to the employee's supervisor typically by invoking an email application to send an email. After this action, a decision node is placed to enable the supervisor to either approve or reject the leave. The decision will determine what path the workflow will take. A client application

might provide a user interface to allow the supervisor to input this decision. If the leave is rejected, then the workflow will notify the employee and terminate the workflow process. If the leave is approved, another workflow path will be taken that will register the leave on the Human Resources (HR) and leave management system simultaneously. Thereafter, the workflow will notify the employee that the leave has been approved and terminate the workflow process.

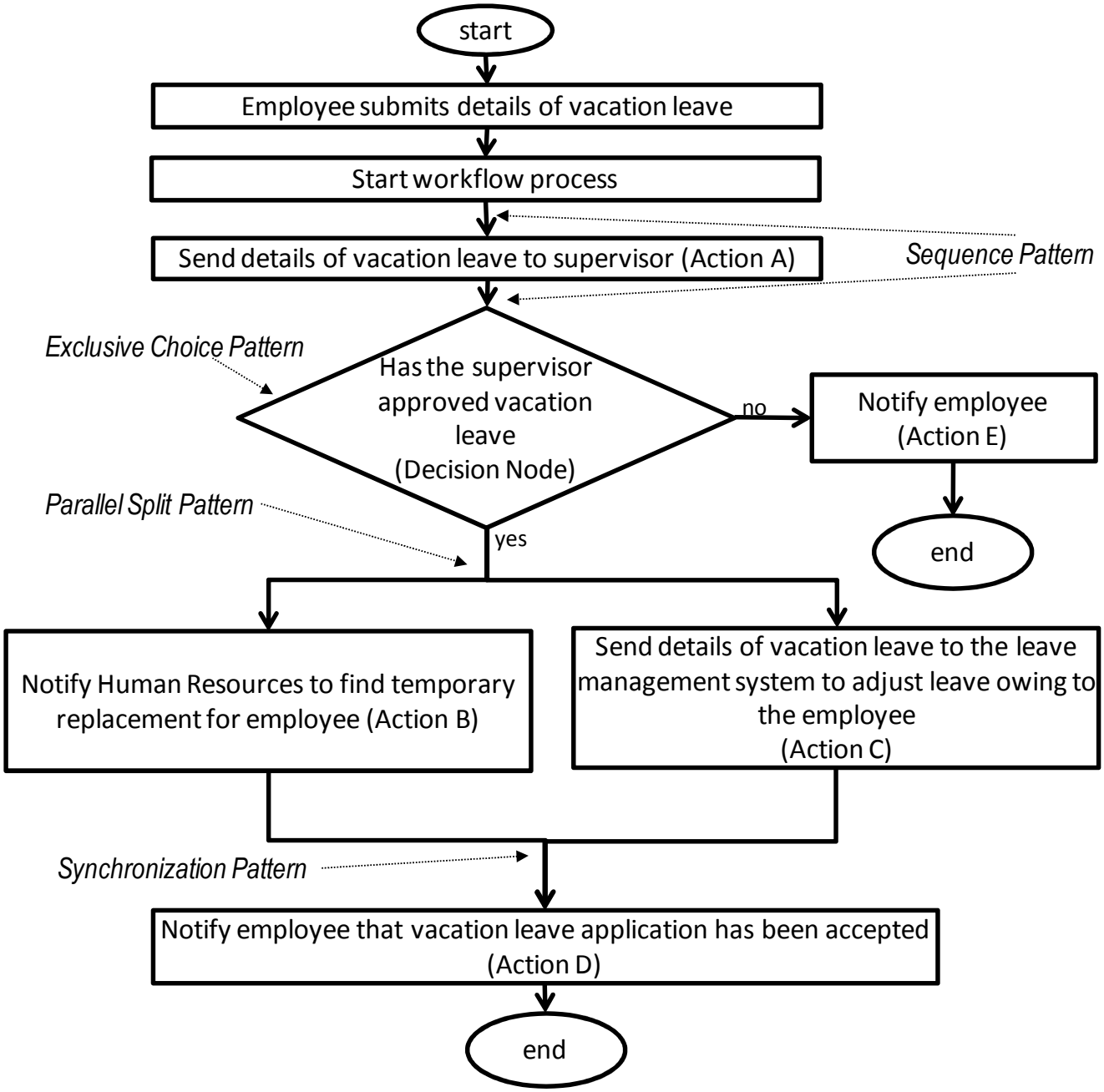

Figure 12: A workflow flowchart for an employee requesting vacation leave.

The workflow example can also be seen through the lens of workflow patterns. The workflow example starts with the sequence workflow pattern which executes workflow action A followed by the exclusive choice pattern. Based

on a decision made by the supervisor, the exclusive pattern stipulates that only one of the two branches will be executed. If the leave is rejected, action E is executed and the workflow is terminated. If the leave is approved, actions B and C are both executed simultaneously within their own threads of control described by the parallel split pattern. Once both actions have been completed, the synchronization pattern joins both threads of control into one thread of control that executes action D and thereafter the workflow is terminated.

It can be seen that the workflow process carries workflow process data and process rules that can allow the workflow engine to query process data to determine what action to do next. For instance, the approval or rejection of vacation leave by the supervisor cannot be done without first sending the details to the supervisor about the vacation leave (Action A). Once the details of the vacation leave have been sent to the supervisor, an indicator or marker to show that this action has been executed will be saved within the process data. The addition of this indicator or marker inside the workflow process data will enable the workflow engine to carry out the next action [25].

The workflow engine invokes client applications to initiate actions within the workflow process. The client application might require information to execute the action. For instance, the workflow engine will invoke an emailing application to send details of the vacation leave to the supervisor. The emailing application will require the email address of the supervisor and the content of the email which contains the details of the employee's vacation leave. All of this information is retrieved from the workflow process data and packaged inside an input data container to be used as an input for the emailing application. Conversely, a client application can also output data using an output process data container that will be stored in the workflow process data. For instance, a client application will be used to capture the details of the vacation leave that was inserted by the employee. This captured data will be injected into the workflow process data that will be used by subsequent actions. Regardless, most actions will use or produce action markers that will be used by process rules to orchestrate the workflow process.

In addition to the process data, the workflow engine will also use the process rules to carry out actions. The straight connector (sequence workflow pattern) that connects Action A to the decision node in Figure 12 is essentially a rule that dictates that until Action A is done (an indicator or marker for Action A is put into the process data to signify the completion of Action A), the decision node cannot accept any decisions from the supervisor.

The decision node (exclusive choice workflow pattern) monitors whether or not the supervisor has approved or rejected the vacation leave. As soon as the supervisor approves the vacation leave, a variable or marker is inserted into the process data to indicate this decision. A rule is present that executes Action B and Action C simultaneously (parallel split pattern) or in parallel after the 'yes' decision is recorded in the process data. Thereafter, Action D has to be executed only after both Action B and Action C (synchronization workflow pattern) have been completed. The completion of Action D sparks an end of this workflow process.

In summary, the main components in order to run a workflow process are the workflow process rules, data and actions.

2.8 Workflow Modelling Techniques and Engine Implementations

A few popular workflow engines were evaluated. eZ Components is a fully fledged PHP workflow engine that utilizes data abstraction and object-orientated organization [29]. ActiveBPEL is a Java workflow engine built using the principals of Service Oriented Architecture (SOA) although the architecture of the actual engine is complex combining aspects of layered, event-driven and client-server architecture [30]. Another Java workflow engine called Activiti uses the Process Virtual Machine (PVM) architecture although its core engine uses elements of object-oriented organization, client-server and event-driven architecture [31]. jBPM5 is business process management workflow application for running business workflows that contains aspects of embedded, layered and event-driven architecture [32]. Workflow engines are complex software applications and therefore most workflow engines that were scrutinized did not have single pure architectures but rather consisted of a heterogeneous architecture. Furthermore, none of these workflow engines take the blackboard approach.

2.8.1 Workflow Modelling and Languages

Before any workflow is created inside a workflow management system, the workflow is modelled based on the requirements of the business or scientific process to be automated using any preferred modelling technique, some of which have already been mentioned in Section 2.3. Additionally, any further refinements and optimizations can be implemented in the workflow model of the business or scientific process. Thereafter, the workflow model can be created inside the workflow management system. This is where workflow languages are used [33]. These workflow languages are used by workflow designers to specify workflows that can also be understood by the workflow management system. Workflow languages are domain specific; they are geared towards specific needs of WFMSs so much so that some WFMSs have their own specific workflow language. As an example, IBM's FlowMark workflow management system has a graph based workflow language called FlowMark Definition Language (FDL) which is specifically designed and used only by that workflow management system [34]. However, some workflow languages are expressly developed for standardization purposes so they can be used by many workflow management systems. Business Process Execution Language (BPEL) is a prime example where leading industry vendors namely IBM, Microsoft, Oracle, BEA Systems, SAP and Siebel Systems came together to develop this language for standardization purposes that is widely used within many of their products in the realm of workflow management [35].

Workflow languages can be classified by the methodologies that they use and by the underlying workflow modelling techniques used. Some popular workflow languages have been classified by [36] as being graph-, Petri-nets, UML activity- and script-based. The type of workflow language and the underlying modelling techniques used

sometimes hints to how a workflow engine has been implemented. The workflow language and underlying modelling techniques used within a workflow engine also affects the amount of workflow and data patterns that can be implemented.

2.8.2 Graph-Based Workflow Management Systems

Many workflow management systems use graph based workflow languages. IBM's FlowMark was one of the first workflow management systems released into the market. The workflow designer component of this WFMS allows workflow designers to create directed acyclic graphs that capture the flow of the business process to be automated [34]. Another workflow engine that does the same is FLOWer [37]. It consists of a fully fledged workflow designer called FLOWer studio that allows workflow designers to define the flow of business processes by creating workflow graphs with forms that can be used to enter and show data within the business process and role definitions for workflow activities that delineate the level of authorization of different users to these workflow activities.

eZ Components is an open source fully fledged PHP WFMS that uses a graph based workflow language for defining workflows [29]. The workflow management system's workflow engine source code was scrutinized. The builders of this workflow engine start from a legitimate supposition that a workflow process definition is activity based. The workflow process definition is essentially a set of activities connected together by transitions. The workflow activities are mapped to nodes and transitions are mapped to edges of a directed acyclic graph. Every single workflow has a start node, workflow activity nodes and many end nodes depending on the kind of workflow connected together by directed edges. To create a workflow inside this WFMS, workflow designers use its graph based workflow definition tool. When a new instance of that workflow is created, the workflow engine actually creates a directed acyclic graph that represents the workflow in memory. To orchestrate this workflow instance, an entity called the task manager traverses through this graph embodying the workflow process using a level-order traversal strategy as the workflow process instance progresses step by step. The task manager keeps track of the current workflow activity or activities (node/s) to be executed. To determine what activity/activities that are required to be executed next, the task manager simply queries the immediate nodes/s (activity/activities) that are connected to the edges emerging from the current node/s (level order transversal strategy) and jumps to these nodes once the current nodes have been executed. The task manager transverses through the graph in this fashion executing workflow activities until an end node is reached thereby terminating the workflow.

Workflow engines that use the principals of graph based modelling to orchestrate workflow processes have one general limitation. These kinds of workflow engines have difficulty executing workflows with some kind of looping and repetition of workflow activities. This limitation lies in the use of the underlying data structure (the Directed Acyclic Graph) that is used to orchestrate workflows. The DAG has a restriction that an executed or visited node cannot be reached and visited again. This means that a workflow process cannot jump back to a previous point to repeat a certain part of a workflow process.

Four out of the twenty main workflow patterns as defined in [14] and [24] capture some kind of looping or repetition of workflow activities. eZ Components can only support one of the four looping patterns [29]. This also applies to IBM's FlowMark [34]. FLOWer is an exception to this case as it can support three out of the four looping patterns [14]. The FLOWer workflow engine allows the creation of nested workflow graphs where a node can be further developed into sub-graphs or sub-workflows. A set of workflow activities can be repeated if the set is made into a sub-graph within a node. Without breaking the underlying design precepts, this workflow engine can achieve looping by repeatedly executing that single node that is composed of the sub-workflow to repeatedly execute the set of workflow activities.

2.8.3 BPEL and Workflow Engine Implementations

Service Oriented Architecture (SOA) is based on leveraging the use of services such as RESTful and SOAP web services to build applications and business processes. Business Process Execution Language (BPEL) is specifically designed to be used as a standard within the SOA environment to build and make use of services. BPEL is an XML-based language that allows workflow designers to create business processes by defining the exact order in which services are to be invoked. In conforming to SOA, workflow processes within BPEL do not execute workflow activities directly but invoke services that will then execute those activities [38].

Business processes or workflows built using BPEL can be run by BPEL workflow engines. There are many open-source and commercial workflow engines that faithfully comply with and can run business workflows developed in BPEL. According to [24], BPEL supports 62.5% of the main workflow patterns. BPEL workflow engines might not fully comply with the BPEL standard and might support less workflow patterns than BPEL. ActiveBPEL has 60% workflow patterns support [30]. Silver is a BPEL workflow engine that a designed for embedded systems and supports 57.5% of the main workflow patterns [39].

It can be cumbersome for workflow designers to create business processes in BPEL by directly developing the XML document that captures the business process. Therefore, many workflow management systems have created workflow designer components that allow workflow designers to create a graphical representation of the business process through a user friendly GUI which creates the BPEL process in XML in the background. Apache ODE is an open source BPEL workflow engine that consist of a designer plug-in that runs inside the Eclipse Integrated Development Environment (IDE) [40]. Oracle has provided a free user friendly IDE called jDeveloper that allows the creation and deployment of BPEL workflow processes to their WFMSs [41]. This demonstrates that no matter what design approach, workflow language and architecture any workflow management system uses, a user friendly GUI that allows workflow designers to build their workflows can always be implemented that abstracts away any complexities.

There are many BPEL workflow engines which are commercially available. Silver and BPEL-Mora are workflow engines specifically designed for embedded systems. Their implementation details are provided in [39] and [42]

respectively. ActiveBPEL is the most popular open-source BPEL workflow engine and its architecture and implementation details are provided in [43]. An analysis on many BPEL workflow engines has been made by [44] and they have provided a general internal architecture that most BPEL workflow engines share. Most BPEL workflow engines follow that same deployment process as shown in Figure 13.

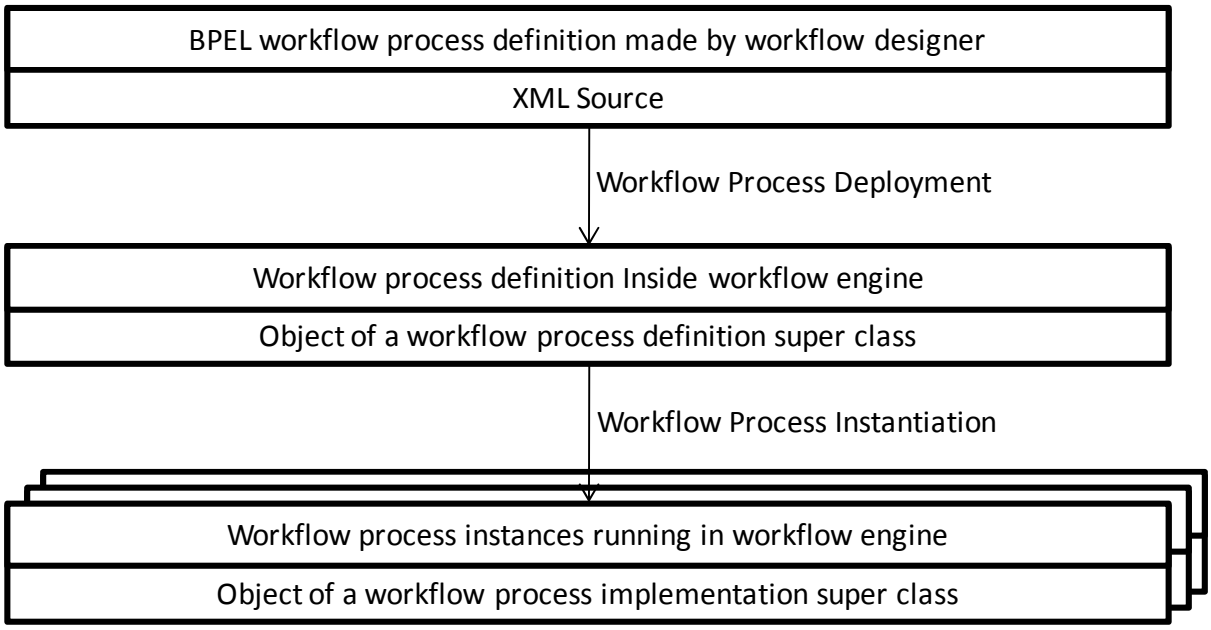

Figure 13: Deployment process for most BPEL workflow engines.

A BPEL workflow process is developed by a workflow designer. BPEL has constructs in XML such as `<sequence>`, `<pick>`, `<switch>` and `<while>` that define the flow of the business process. Each BPEL construct has a matching definition class within the BPEL workflow engine that can be instantiated to capture the functionality of the construct. When a BPEL process is deployed to the workflow engine, the BPEL XML document is parsed by the workflow engine. The order of the BPEL constructs which defines the workflow process is parsed into objects of relevant construct definition classes. A workflow definition super class is instantiated that is composed of a list of these construct definition classes and any other information for running this workflow from the BPEL XML source. Each BPEL construct definition class has a matching construct implementation class that has knowledge on how to execute the construct when a workflow process is running. When a workflow process is instantiated, the workflow engine will use the objects of workflow definition classes in conjunction with the matching objects of the workflow implementation classes to run the workflow [44]. In summary, BPEL workflow engines are able to load BPEL XML source code to create internal workflow process definitions. The process definitions will be used to instantiate workflow process instances.

BPEL is a block-structured language. It consists of constructs or blocks such as sequence, if-then-elseif-else, while and pick to define workflow processes. The block-structured nature of this language prevents the implementation of some workflow patterns namely the multi-merge and arbitrary cycles workflow patterns. The multi-merge pattern

(see Section 2.4 for explanation of this pattern) is unsupported since it is impossible for multiple threads of execution to run through the same path (multiple threads cannot enter the same block) in a workflow process [45]. The arbitrary cycles pattern encapsulates an unstructured loop that can have multiple entry and exit points. Block-structured languages such as BPEL cannot implement this pattern as they do not have to ability to arbitrarily jump to any past or future point in a workflow process [46]. Furthermore, some BPEL constructs simply have not been developed to support some workflow patterns such as the structured discriminator pattern and certain patterns that involve different kinds of looping [25].

2.8.4 Petri-Net Based Workflow Engine Implementations

Yet Another Workflow Language (YAWL) is another XML based workflow language similar to BPEL that is specifically focused on maximum support of workflow patterns [47]. The creators of the workflow patterns initiative advocate that Petri-nets should be used to model workflow process since it is able to support most of the defined workflow patterns [48]. Therefore, the designers of YAWL developed this language using the underlying principals found in Petri-nets. YAWL supports 95% of all the main workflow patterns [49]. The only workflow pattern that is not supported by YAWL is the implicit termination pattern. This pattern encapsulates a situation where a workflow process is terminated when there are no more active workflow activities left to be executed in the workflow process. Petri-nets however expressly require the definition of a final node to terminate workflow processes. The designers of YAWL offer a solution to partially support this pattern where multiple end nodes can be joined to a final end node using the synchronizing merge pattern (see Section 6.2.1 for details about Synchronizing Merge Pattern). However, they have not implemented this solution to force workflow designers to create workflow processes that have explicit successful completion paths to prevent any possible deadlocks in the workflow process [50].

YAWL has currently only one workflow engine implementation. Its internal architecture and implementation details are provided in [51]. Upon inspection of the source code, the general architecture was found to be similar to that of BPEL workflow engines. Like BPEL workflow engines, the YAWL workflow engine also loads YAWL XML source code that defines a workflow process to create internal workflow process definitions. Each YAWL workflow construct in XML has a matching workflow construct definition class. However, YAWL is based on Petri-nets. The workflow construct definition classes capture each workflow construct using Petri-nets.

Besides YAWL, there are only a few other workflow engines that use the Petri-nets formalism [52]. Grid Workflow Execution Service (GWES) is another Petri-nets based workflow engine that is designed to work in grid environments. Its internal architecture and implementations are provided in [53].

2.8.5 Script based Work Languages and Workflow Engine Implementations

Script-based workflow languages are high level languages that are designed to directly specify workflow constructs used in workflow processes. Engines that employ script-based workflow languages are typically used in software

development environments that need more control over the workflow based applications that they are developing. A hallmark of script-based workflow languages is that they are extensible and software developers are unfettered in adding more workflow control constructs that are not already available in the language. However, such workflow systems are not user friendly. Workflow designers cannot define workflows in simple to use UIs but require a software development background to define workflows by directly developing them inside the workflow language [54].

Mobile is one of the first workflow managements based on its own script-based language Mobile Script Language [55]. The system including its language was developed in Java. Since the Java code has direct access to the state and flow of workflow processes, any workflow pattern can theoretically be implemented if the Mobile Script Language was further extended. It also consists of a rudimentary workflow editor called MoMo to allow workflow designers to create workflow processes using the workflow constructs provided by default. The workflow example provided in Section 2.7 of employees requesting for vacation leave can be developed in the Mobile Script Language shown is code listing 1. Due to the high level nature of script-based workflow languages, it is simpler and quicker to define workflows as opposed to using traditional software languages such as Java or C++.

```
WORFLOW_TYPE EmployeeVacationLeave
     WORKFLOW_DATA
           LeaveInfo: li,
           SupervisorResponse: sr
     END_WORKFLOW_DATA

     CONTROL_FLOW
           sequence(SendDetailsToSupervisor,
                 sequence(AssessLeave),
                 ifthen(sr.approved == true,
                       split(NotifyHR,AdjustLeaveInManagementSystem),
                       merge(NotifyApprovalAccepted),
           sequence(NotifyApprovalRejected)))
     END_CONTROL_FLOW
END_WORKFLOW_TYPE
```

Listing 1: A workflow for an employee requesting vacation leave written in the Mobile Script Language.

GWENDIA workflow is another workflow management system specifically designed for scientific workflows which require large amounts of data and data sets to be utilized and manipulated. It uses GWENDIA-script or gscript for defining workflows. Its internal architecture and implementation details can be found in [56].

It is difficult to gauge the level of workflow pattern support for script-based workflow engines since the support for workflow patterns is not fixed. Most of these script-based workflow management systems are designed to be extended to implement any unsupported workflow patterns.

2.8.6 General Analysis and Observations

The previous sections on different workflow engines and their internal implementation details demonstrate that every workflow engine has its own distinctive internal architectural style and implementation of the various underlying workflow modelling techniques. Some workflow engines solely use DAGs or Petri-Nets as their starting point to build a workflow engine while others use a combination of many different modelling techniques. Some workflow engines are designed to work with a standardized workflow language while others use their own unique domain specific workflow language. All of the workflow engines that were investigated are not built using any one single architectural pattern. A group of pure architectural patterns are amalgamated into a heterogeneous architecture for these complex software applications. Furthermore, the architecture and underlying modelling techniques used in these workflow engines can affect the number of workflow patterns that can be supported.

Many workflow engines investigated have an entity (can be called a task master) to keep track of the transitions between the activities in order to know what activities to execute next. The eZ Components workflow engine is one representative example of this. This workflow engine maps workflow definitions to DAGs in order to run workflow processes as explained in Section 2.8.2. To run these workflow processes, an entity traverses through these DAGs. There is no need to know about the state of the graph or how far the graph has been traversed. The only knowledge that is required is what are the current activities to be executed and the transitions to the next set of activities to be executed. Similarly, workflow engines that run BPEL processes also keep track of the current set of activities to be executed and the transitions to be next set. BPEL is block-structured so an entity keeps track of the current blocks of the BPEL process and the transitions to the next set of blocks.

The state of the workflow process and its data play no role in the orchestration of workflows in these types of workflow engines. The only knowledge is needed is the current activities to execute and the transitions to jump to the next set of activities to execute.

Indeed most workflow engines have this salient characteristic. Recall, Van der Aalst in both [14] and [24] conducted a study to determine how many workflow patterns the current leading workflow management systems could implement. Van der Aalst noted that most of these workflow management systems abstracted from states. The workflow process state and data are not explicitly modelled and factored into these workflow management systems which prevented the implementation of some complex workflow patterns ([14] and [24]).

2.9 Summary

This chapter fleshes out some background information that is relevant to the research presented in this dissertation. The components, modelling techniques and behaviour of workflow processes and the many ways they are executed within many different workflow engines are explained. It is important to bear in mind that a workflow process can be decomposed into workflow process rules, process data and process actions. The next chapter will introduce the blackboard paradigm and use these three workflow process components and their behaviour to build a workflow engine utilizing the blackboard paradigm.

3 Using the Blackboard Paradigm in the Context of a Workflow Engine

3.1 Introduction

This chapter uses the theoretical concepts surrounding workflows discussed in the previous chapter and maps them to the blackboard paradigm. The blackboard architecture is discussed in detail with each component in this architecture and its behaviour being examined. Petri-nets are used to conceptualize the behaviour of the internal components of a workflow process which is mapped to the different components of the blackboard paradigm. A general discussion on the advantages and disadvantages of using a blackboard paradigm to build a workflow engine is provided. Previous research done on the subject of the feasibility of building a workflow engine using the blackboard paradigm is investigated and the solutions proposed in existing research are compared with the solution proposed in this dissertation.

3.2 Blackboard Paradigm

A problem solving model entails the organization of reasoning steps and knowledge to construct a solution to a problem. There are several types of problem solving models. For instance, there is the backward reasoning model where the problem is solved reasoning backwards from the objective to be achieved towards an initial state. Conversely, the forward reasoning model has the problem solving logic steps from an initial state towards an objective [57]. Traditional workflow engines can be said to loosely adhere to the forward reasoning model. In an opportunistic reasoning model, a problem can be solved by applying pieces of knowledge either forward or backward at the most opportune time. Ultimately, problem solving models address the fundamental issue of the organization of knowledge and an approach to applying that knowledge to solve a problem [58].

The blackboard paradigm is used as a flexible problem solving approach and it utilizes the opportunistic problem solving model. Blackboard architectures support multiple problem solving techniques, multilevel abstraction of situations and control processes and responsive opportunistic control of workflow process activity [59].

Figure 1 depicts a simple view of the blackboard architecture. From Figure 1, it can be seen that the blackboard paradigm consists of two kinds of components namely a central data structure that embodies the current state (blackboard) and independent components (specialists) that operate on the blackboard. The next two sections will consider the specialists and blackboard in more detail.

3.2.1 Specialists

Specialists consist of two major components namely the condition and action. The condition component exists to initiate when the specialist can contribute to the blackboard. Also known as a precondition component, it stipulates when a specialist should execute its actions and more specifically which set of actions from the action component to execute contingent on what conditions in the condition component are satisfied. The action component consists of procedures to contribute to the blackboard. The procedures being executed insert new data or alter existing data on the blackboard. These changes made to the blackboard are explicit and can be understood by other specialists [60].

Specialists are autonomous components that require no assistance nor even need to acknowledge the existence of other specialists. However, the specialist must understand the existing information and contribute information to the blackboard that can be comprehended and used by other specialists.

3.2.2 The Blackboard

The blackboard is the global data structure that is monitored by specialists. The data could consist of raw data added by specialists and partial solutions to problems. Since specialists do not intercommunicate, the blackboard is also serves as a communication medium between the specialists. The blackboard is also important for specialist triggering. The very data on the blackboard can be used as a triggering mechanism for specialists [16].

Revisiting the blackboard metaphor, as more specialists contribute to the blackboard, the information on the blackboard grows making it harder for specialists to monitor or to look up pertinent information on the blackboard. A common way that this problem is solved is to divide the blackboard into zones that contain specific types of information [16]. Furthermore, each zone has some sort of positioning metrics for efficient retrieval of information. For instance, information can be arranged numerically, alphabetically, using key-value pairs, by importance, by last modified etc. There can also be dynamic or static regions on the blackboard where information that should not be changed should be put into the static region and information that can be changed can be put into the dynamic regions [61].

The blackboard does not restrict any type of information that can be put on it. This introduces a problem of how to represent information on the blackboard. From the blackboard metaphor, human specialists can doodle on the blackboard when trying to contribute to a problem. A specialist can add anything from pictures, lists, tables, equations etc. It is important that other specialists can understand what a particular specialist has added to the

blackboard. Since the blackboard does not restrict what type of information can be put on the blackboard, specialists have to interact with the blackboard responsibly and not leave incompatible information that may be misunderstood by other specialists. A common understanding of information on the blackboard information must be maintained. However, it might be desirable if a certain set of specialists have their own special jargon or a different language that cannot be understood by other specialists so only a single set of specialists can interact on the blackboard. Depending on the application, a trade-off between generalized and specialized representation of the blackboard has to be made by the application developer [62].

3.2.3 Need for Control

Opportunistic problem solving presents its own set of challenges. A circumstance can arise where two or more specialists operate on the same blackboard data at the same time which might comprise the integrity (data corruption) or security (read or write privileges) of this global data structure. Reverting back to the blackboard metaphor, a situation may arise of many human specialists responding to a particular change on the blackboard and all scuffle to get to the blackboard in a bid to provide their input. Civility must be established and human specialists should contribute to the blackboard one after the other in an orderly fashion. A simple control approach to accomplish this is to have only one piece of chalk. It can be given to only one human specialist at one time that can be passed around. This inevitably introduces the necessary control component back into the blackboard paradigm [63].

The control component is distinct to the individual specialists in the blackboard paradigm. It is responsible for synchronizing the specialists' access to alter the blackboard. It can also monitor and alter the blackboard. It is imperative that the control component remains distinct to the specialists and is unable to assume control over any part of the specialists (execute the conditions or actions of the condition-action pairs of specialists). The control component should not be able to invoke specialist rules (condition-action pairs) or any other part of specialists. If this is neglected, the modularity of the specialists (independent components) will be forfeited thereby totally comprising the opportunistic problem solving model provided by the blackboard paradigm.

Depending on the application, the complexity of the control component can vary significantly [64]. Blackboard systems can support diverse types of control components or strategies. Application developers are unfettered to create any type of opportunistic control technique depending on the requirements of the application [64].

Generally in the literature that has been reviewed and the blackboard applications that have been investigated, this control component is simply termed as the 'controller'. This term is deceptive as it leads one to believe that the controller completely takes possession of the specialists and turns them into passive drones submitting to the controller's bidding. The author feels that the term 'orchestrator' better defines this control component. Just like a musician (orchestrator) who arranges voices and instruments for a musical performance, the control component in

the blackboard system arranges blackboard access for specialists to achieve an overall goal or to solve a problem. The complete view with all the components of the blackboard paradigm is shown in Figure 14 [65].

Simple control components employ a "first come first serve" strategy. The first or the swiftest specialist that requests access to the blackboard will be given access. In applications where specialists operate on the blackboard for long periods of time, the control component might consist of a queue where other specialists that require access to the blackboard will be queued that later can operate on the blackboard once the active specialist has completed its contribution to the blackboard. HASP/SIAP was a software system developed using the blackboard paradigm to interpret sonar signals emitted by the moving crafts in an ocean region [66]. The control component of HASP/SIAP employed this strategy where a change in the blackboard would occur which then prompted the control component to permit the specialists to access the blackboard in FIFO order [66].

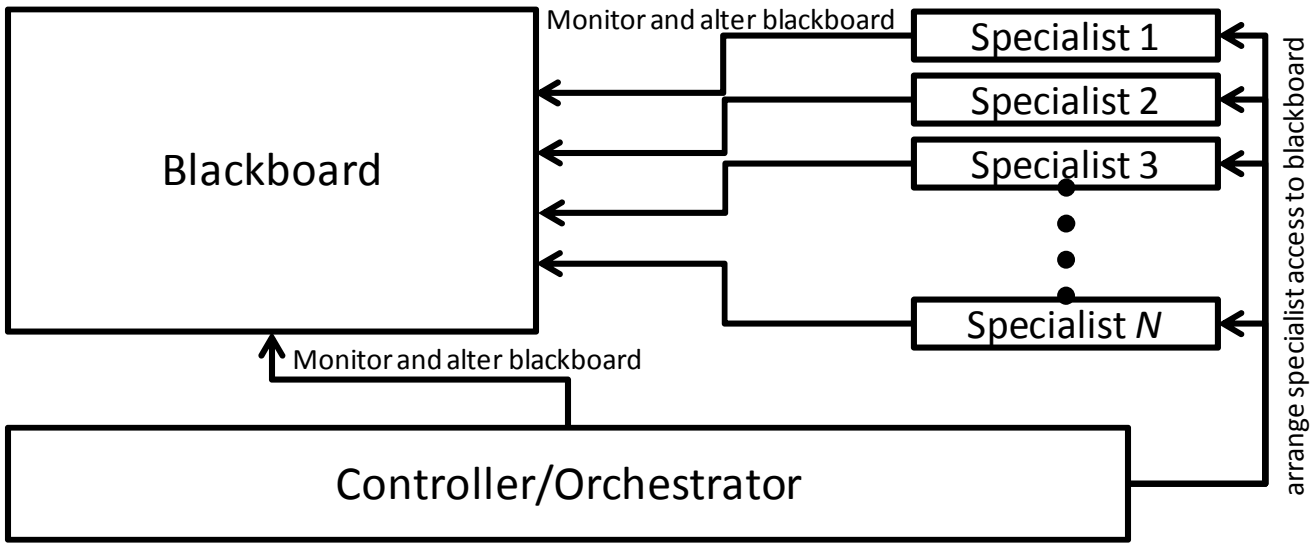

Figure 14: The complete view of the blackboard paradigm adapted from [65].

More advanced control components consider the most suitable specialist to contribute to the blackboard instead of the first specialist in a queue. The advanced control component is in charge of the course of problem solving and ensures that important aspects of the problem are being focused on. The control component has to monitor the blackboard and depending on the information on the blackboard select a course of problem solving. Thereafter, the control component has to evaluate the level of contribution to be made by specialists that request access to the blackboard simultaneously. To keep problem solving on track, the control component will select the most appropriate specialist. A simple example would be a control component requesting the computation times from the pending specialists wanting access to the blackboard and select the specialist with the least costly computation and the one that has not made its contribution to the blackboard yet [67]. Additionally, the control component can also add the decisions that were made by it to the blackboard that can later be used to help decision making in the future. BB1 is a generic blackboard framework that uses this type of control strategy [68].

3.2.4 Event-Based Blackboard Systems

From the blackboard metaphor, specialists do not interact with each other and can only have an opportunity to apply their contribution to the blackboard when a change on the blackboard occurs. Specialists can only act when a blackboard event occurs which entails information being added, altered or removed from the blackboard. Rather than specialists ceaselessly searching the blackboard for changes, which is inefficient, specialists can inform the blackboard system of what types of information they are interested in. This is recorded in the blackboard system. When the blackboard is altered, the blackboard system will consider which specialist to inform. Specialist can also respond to other external events such as timing events. Event-based activation of specialists use less software resources and improves efficiency [65].

In addition to specialists being composed of condition-action pairs (rules), specialists can also have triggering information although this is not necessarily required. The triggering information can be sent by the specialist when it is loaded into the blackboard system to the controller/orchestrator to be recorded. The controller/orchestrator can monitor the blackboard for changes. If a change occurs, the orchestrator can check if any of the triggering information is satisfied thereby informing the corresponding specialist to act [69]. This tallies in with event-driven architecture which essentially consists of event emitters that are responsible for detecting events and event consumers that are responsible for reacting to events [70]. An event is considered to be a change on the blackboard. The controller/orchestrator can be regarded as an event emitter and the specialists can be regarded as event consumers.

It is important to realize that the triggering of a specialist (the controller/orchestrator informs the specialist to act) does not necessarily mean that the specialist will automatically make a contribution to the blackboard. Triggering simply notifies a specialist to start scanning the blackboard and check if its conditions are satisfied before executing its actions upon the blackboard. This is the approach taken in BB1 [68].

Other systems such as HASP/SIAP have specialists actively scanning and interacting with specific or important zones of the blackboard while sending triggering information to the controller/orchestrator regarding changes to less important zones to the blackboard. HASP/SIAP consisted of a blackboard where raw input data was posted regularly on the lower levels of the blackboard. Thereafter, this raw input data was used by the specialists to formulate a solution in the upper levels of the blackboard. Specialists were designed to only perform problem solving on the upper levels of the blackboard. If new information was posted to the lower levels of the blackboard, using the triggering information provided by the specialists, the controller/orchestrator triggers the relevant specialists that will then scan the relevant lower zones of the blackboard to apply their expertise and copy data to the upper levels of the blackboard for further problem solving. Besides blackboard events, specialists also responded to timing or clock events. The HASP/SIAP controller/orchestrator also consists of a clock events component that triggers specialists using the system clock in order to synchronize the blackboard with the real world and time-based calculations [66].

Some systems such as ARBS do not even consist of a controller and only consist of different kinds of specialists constantly monitoring the blackboard. ARBS is a blackboard system that is used to control plasma processing units [71]. It consists of a small blackboard without zones that consists of only data required for performing calculations for plasma deposition control. All the different kinds of specialists of ARBS are able to constantly monitor the entire blackboard and perform calculations on the data available on the blackboard. Only one specialist can change the blackboard at one time. The blackboard itself has a locking mechanism that is used to synchronize access to specialists. The blackboard becomes locked when any single specialist is altering the blackboard and access to the blackboard is granted to only a single specialist at any one time in FIFO order therefore eliminating the need for a controller/orchestrator.

Depending on the application, event based triggering of specialists can be implemented to a certain degree or completely discarded. In some cases, designing specialists to be active entities constantly scanning the entire blackboard for changes is inefficient especially when specialists only have the opportunity to interact within the blackboard when a blackboard event happens. Rather, specialists should only start scanning the blackboard when a change on the blackboard has actually occurred.

3.2.5 External Data Insertion on the Blackboard

External data can always be put on the blackboard at any time. Typically, external data can only be put inside one zone on the blackboard so as not to disturb other regions of the blackboard where active problem solving might be occurring. While specialists are monitoring and engaging with a problem on a specific zone of the blackboard, new data can put on another zone of the blackboard by an external party that might be useful to solve the current problem. If the specialists are not notified of this new information, problem solving might be adversely affected as potentially useful information that might help solve the problem will be left unnoticed.

The external party has to inform the orchestrator (controller) that new information has been inserted that will then inform the relevant specialists. If some sort of event based specialist triggering is implemented, when new information has been posted by an external entity, the orchestrator/controller has to be made aware of this insertion so it can trigger the relevant specialists to shift their attention to this newly inserted information. However, this is not needed if event based triggering is not implemented. Specialists will notice new information on the blackboard if they are constantly scanning the entire blackboard for changes [72].

3.2.6 Serial vs. Concurrent Blackboard Systems

The power behind a blackboard system is the ability to conduct opportunistic problem solving using a data-driven control structure and specialists interacting simultaneously on one centralized data structure to solve large complex problems. The ideal blackboard system will solve problems using specialists interacting asynchronously, in parallel on one centralized data structure with no memory disputes (multiple entities altering the same data). However, blackboard system designers realize inherent problems maintaining data and semantic consistency with the

blackboard while implementing concurrent specialist execution on the blackboard. The problem of control commonly ends up being answered by serial implementation of a blackboard system. Serial Blackboards prevent multiple specialists from interacting with the blackboard at the same time. The controller/orchestrator is present within the system that guides the path of problem solving and selects appropriate specialists one-by-one based on some overall problem solving strategy or goal. Proponents of serial blackboards [73, 74, 75 and 76] have outlined several issues facing blackboard designers which are also important in the context of building a workflow engine:

- Implementing speculative parallelism adversely affects performance and might lead to an unstable system that is prone to synchronization problems.
- Memory management and coherence becomes problematic when multiple entities are engaging simultaneously upon the same data.
- Introducing concurrency control by introducing process locking can restrict opportunistic problem solving.
- Workflow engines are real time systems that require predictable response times. Concurrent execution of specialists might lead to unpredictable response times.

However all of these proponents recognize that the introduction of a serial central controller/orchestrator creates a bottleneck (reduces performance) and eliminates specialist concurrent execution.

Depending on the application, specialist concurrency might be required. Specialist concurrency may be required in the context of building workflow engines. This discussion of whether or not a concurrent or a serial implementation is required will resume in Section 3.5 when the blackboard paradigm is introduced in the context of building a workflow engine.

The design and analysis of concurrent blackboard systems is considered in [77]. Here three main types of concurrent blackboard implementation approaches are identified namely the distributed blackboard (DBB) approach, the blackboard server (BBS) approach and the shared memory blackboard (SMBB) approach.

The DBB approach seen in Figure 15(a), consists of separate blackboards for each concurrent specialist execution thread. The separate blackboards contain data created locally by the thread as well as copies of other data from other blackboards that are required by each thread. When some data is required by some thread that is available in some other thread, the node processor for that thread will be able to communicate with the other concurrent threads and retrieve and create a copy of that data. This approach can create data consistency issues with overlapping data (copies of the same data) among the different blackboards. Changes to a single copy have to be relayed to all the other blackboards.

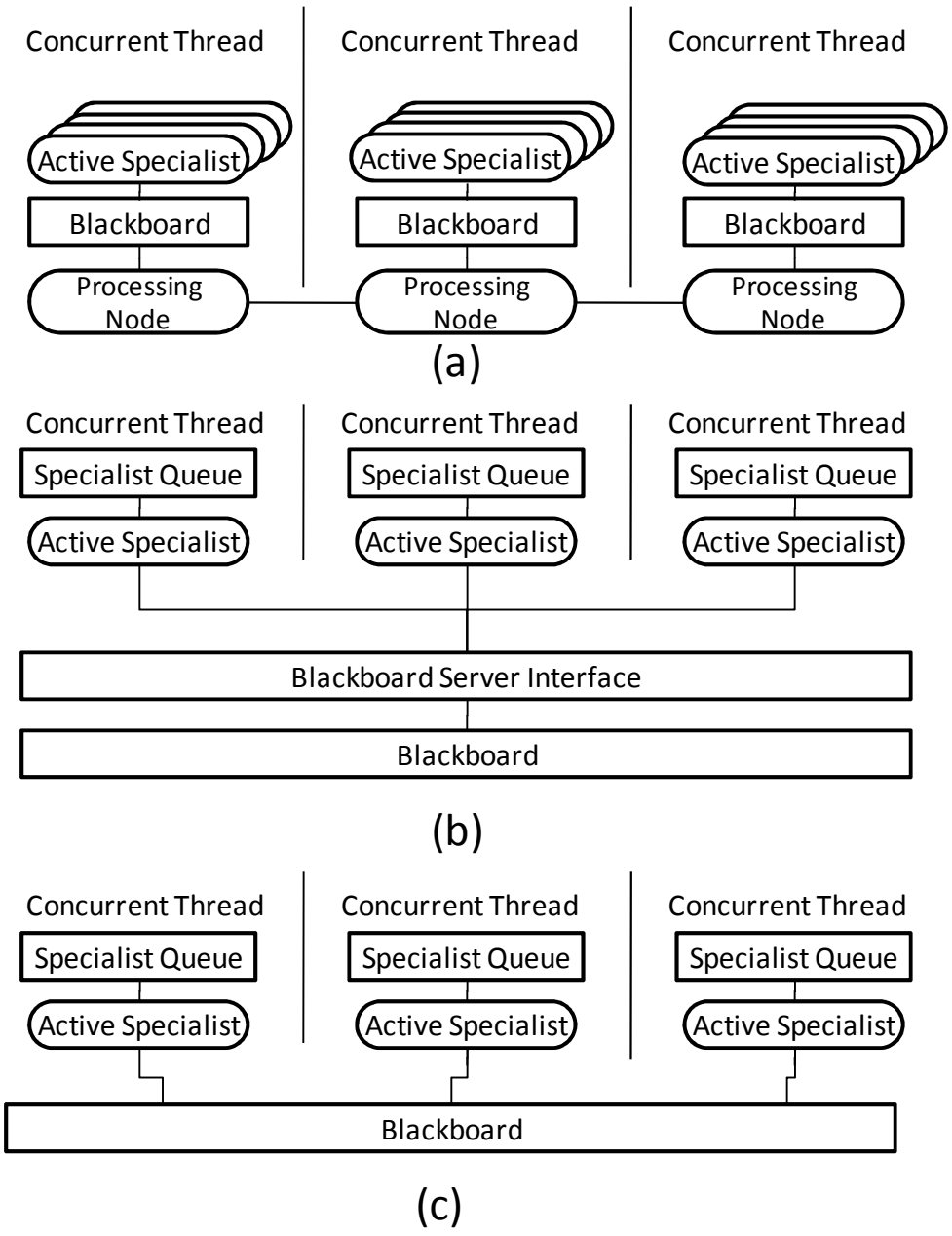

Figure 15: The concurrent blackboard implementation approaches. **(a)** Distributed blackboard approach. **(b)** Blackboard server approach. **(c)** Shared memory blackboard approach.

The BBS approach, shown in Figure 15(b), consists of a single blackboard with specialists interacting via an interface with the blackboard. Unlike the DBB, the BBS does not have multiple copies of data so maintaining data consistency is easy. However, specialists do not interact directly with the blackboard and have to interact with the blackboard interface that manages data consistency. The blackboard interface can employ many data locking strategies such as optimistic or pessimistic locking (to name a few) to maintain data consistency [78]. If two specialists want to update a single piece of data at the same time, the blackboard interface would place a lock on that piece of data so that only one specialist can perform updates at one time. However, multiple specialists can read the same data at the same time (concurrent access) and use this data to perform their tasks. The BBS approach

is effective but introduces another layer to achieve concurrency which reduces performance. It is advised by [77] that the BBS approach should be used for systems with high bandwidth and where specialist-blackboard interaction times are large. This is due to the fact that there is an inherent time delay when specialists connect and execute requests on the server to update the blackboard.

The SMBB approach as seen in Figure 15(c) allows each specialist to directly access the blackboard. This is the most direct approach to give concurrent access to the blackboard to the specialists. Instead of an interface layer that manages data consistency by using some form of data locking, the blackboard itself is created to manage its own data consistency. To accomplish this, the blackboard is split into a set of "buckets" and each bucket is given locking capability. When a specialist is writing to a bucket, it becomes locked to prevent other specialists to write to it. However, multiple specialists can read the bucket at the same time even when it is locked.

It can be argued that no matter what concurrent blackboard implementation is taken, there will always be memory contention on the blackboard and true concurrency can never be realized. This is the limitation of using a centralized data structure as many researchers [73, 74 and 77] have indicated. In [79 – 81], it is suggested that creating specialists that are highly specialised (modular) will solve this problem. Well designed, strongly focused specialists are required to execute or solve single specific tasks. If this design suggestion is used, most memory contentions will be resolved.

3.2.7 Blackboard Design Stipulations or Guidelines

The blackboard paradigm was first used in applications employing complex signal processing for speech and pattern recognition. HEARSAY-II is a speech recognition system that was first developed 1971 using the blackboard pattern [82]. Later, many other software systems were built that employed the principles of blackboard architecture. Nii has published a paper called "Blackboard Systems" in which several of these systems are surveyed [58]. It was noticed that many software systems contain different implementations of the blackboard paradigm. Some systems were built that use some elements of blackboard architecture but lacked crucial implementation aspects of the blackboard paradigm that promote some of the benefits such as flexibility and opportunistic control inherent to the blackboard paradigm. This necessitates some general guidelines and stipulations as to how blackboard systems should be designed and implemented. Barbara Hayes-Roth has provided a framework called the blackboard framework which offers design guidelines suitable for blackboard systems [16]. A few key guidelines and stipulations are listed below:

- The purpose of a specialist is to contribute data to the blackboard that will ultimately lead to the solution of a particular problem. No single specialist should be able to solve a particular problem in one instance.
- The specialist should consist of procedures, set of rules or logic assertions.
- Specialists can only modify data on the blackboard. All modifications should be explicit and visible.

- Specialists should have information about what conditions they can contribute to the solution.
- Specialists are static. The information making up a specialist cannot change.
- Specialists are independent and cannot intercommunicate. Furthermore, a specialist should not require other specialists to make its contribution to the blackboard.
- The blackboard should contain computational and solution state data required by and produced by specialists.
- All information on the blackboard should be explicit and has to be understood and be readily usable by specialists.
- The blackboard can have multiple blackboard zones so the solution state data can be split into multiple hierarchies and represented within these zones.
- Specialists cannot be forced by the controller/orchestrator to contribute to the blackboard.
- The controller/orchestrator can be present to synchronize access of specialists to the blackboard.

The following sections will use the blackboard paradigm in the context of building a workflow engine.

3.3 Petri-Net Conceptualization of a Workflow Process

Petri-nets provide a sound starting point to formulate a foundation for building a workflow engine because according to [48], they provide maximum support for workflow patterns. Petri-nets consist of *transitions* (squares), *arcs* (arrows), *places* (circles) and *tokens* (dots) [21]. From Chapter 2, it is imperative to realize that the entities executing a workflow process in a workflow engine are the workflow process rules, data and actions. These three entities form the workflow process definition that can be modelled into a Petri-net. *Process actions* can be modelled by *transitions*. *Process rules* can be modelled by *places*. Data belonging to a workflow process can be modelled by *tokens*. To demonstrate how a workflow process is modelled as a Petri-net, an example is given illustrated in Figure 16 of a workflow process with a parallel split workflow pattern. After action A is executed, both actions B and C have to be executed. Based on how Petri-nets work, a *transition* can be fired (workflow action can be executed) if the *input place* (workflow rule) contains a *token* (workflow process data) and the data inside the token causes the rules inside the input place to be evaluated to true. This fires the transition (action executed) and the token gets pushed to the output places. In the example in Figure 16, the token inside place P1 will execute action A and the token will be pushed to both places P2 and P3 thereby executing actions B and C.

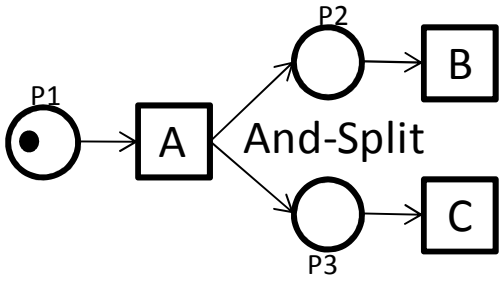

Figure 16: A Petri-net illustrating the Parallel Split workflow pattern.

The way in which a Petri-net is fired can be distilled further into a fundamental relationship that exists between the three entities defined in the workflow process definition which is depicted in Figure 17. A process rule (input place of Petri-net) can monitor the process data (tokens). If process data is available for the process rule to be evaluated to be true (token exists inside input place), the process rule can execute an action (Petri-net transition is fired). Once an action is completed, a marker or variable will be inserted into the process data to reflect the completion of this action (token is pushed to the output place of Petri-net). Thereafter, the next process rule will monitor the process data and the procedure just described will be repeated.

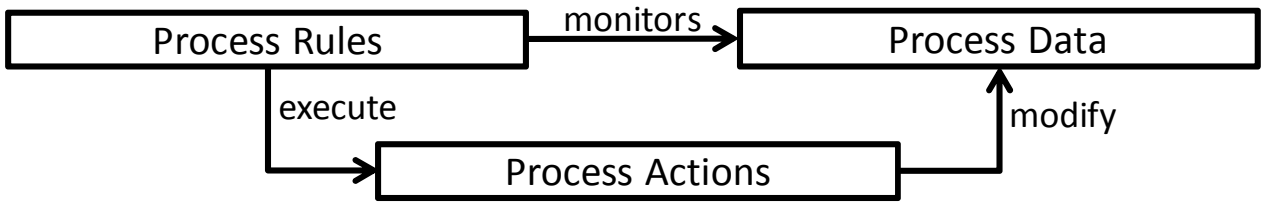

Figure 17: The relationship between process rules, process data and process actions to be executed.

3.4 Mapping a Petri-net Modelling Approach to the Blackboard Paradigm

The fundamental relationship that exists between the components (workflow rules, data and actions) within the workflow process shown in Figure 17 can be easily mapped to the blackboard paradigm. The process data can be put on the blackboard and the specialists that monitor the blackboard can contain process rules and the actions to be executed if the process rules are evaluated to true. Figure 18 shows the relationship between the process data, rules and actions implemented in the blackboard paradigm. Multiple specialists that contain a unique set of process rules and actions to be performed can monitor the blackboard simultaneously. As soon as the process data on the blackboard will allow for a particular specialist's process rules to be satisfied, that specialist will immediately execute its actions. A specialist, as part of its actions, can also alter the process data on the blackboard as so to alter the workflow process's path through the workflow engine. As mentioned earlier in Section 2.10.8, the design of the workflow engine should adhere to the stipulations and guidelines provided by [16].

To simplify the discussion, the introduction of the controller/orchestrator will be delayed and event-based triggering of specialists will be ignored. The purpose of the controller/orchestrator is to synchronize access of specialists to the blackboard. For the purposes of this discussion, it will be assumed that specialists are interacting with the blackboard in an orderly fashion without any competition between them and that they are constantly monitoring the entire blackboard so as to be ready to make a contribution.

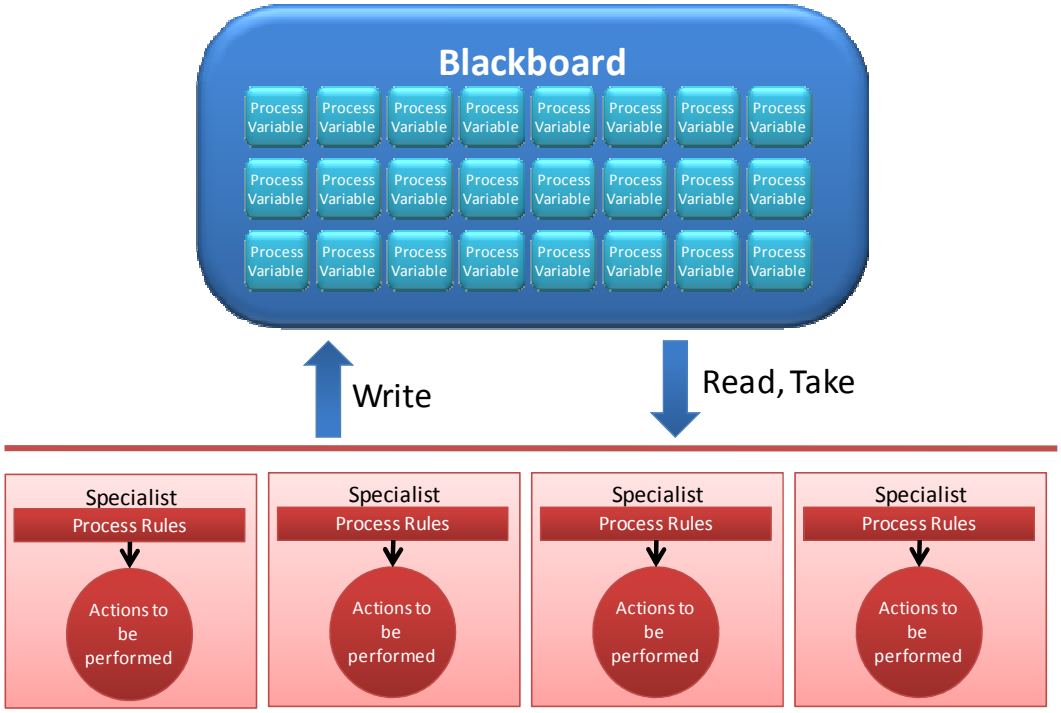

Figure 18: The relationship between process rules, process data and process actions to be executed in a blackboard paradigm.

There are many advantages of using the blackboard paradigm. It is important to note that this paradigm does not make the workflow engine action or task oriented and rigid. It does not require a task master that is aware of each action transition which executes actions manually from one to the next as opposed to many other workflow engines. With the blackboard paradigm, the workflow is designed to execute actions that match the specialist rules. With traditional workflows, once an action or a multiple set of actions are completed, the actions cannot be used or executed again. This is what makes traditional workflow engines linear and rigid. With the blackboard paradigm, this is not the case. A single specialist can execute its actions again and again if the workflow process data on the blackboard allow the process rules to be evaluated to true (opportunistic control). This will make this type of workflow engine more flexible, cyclic and powerful as the workflow designer can create numerous workflow paths. An example of this is illustrated in Figure 19.

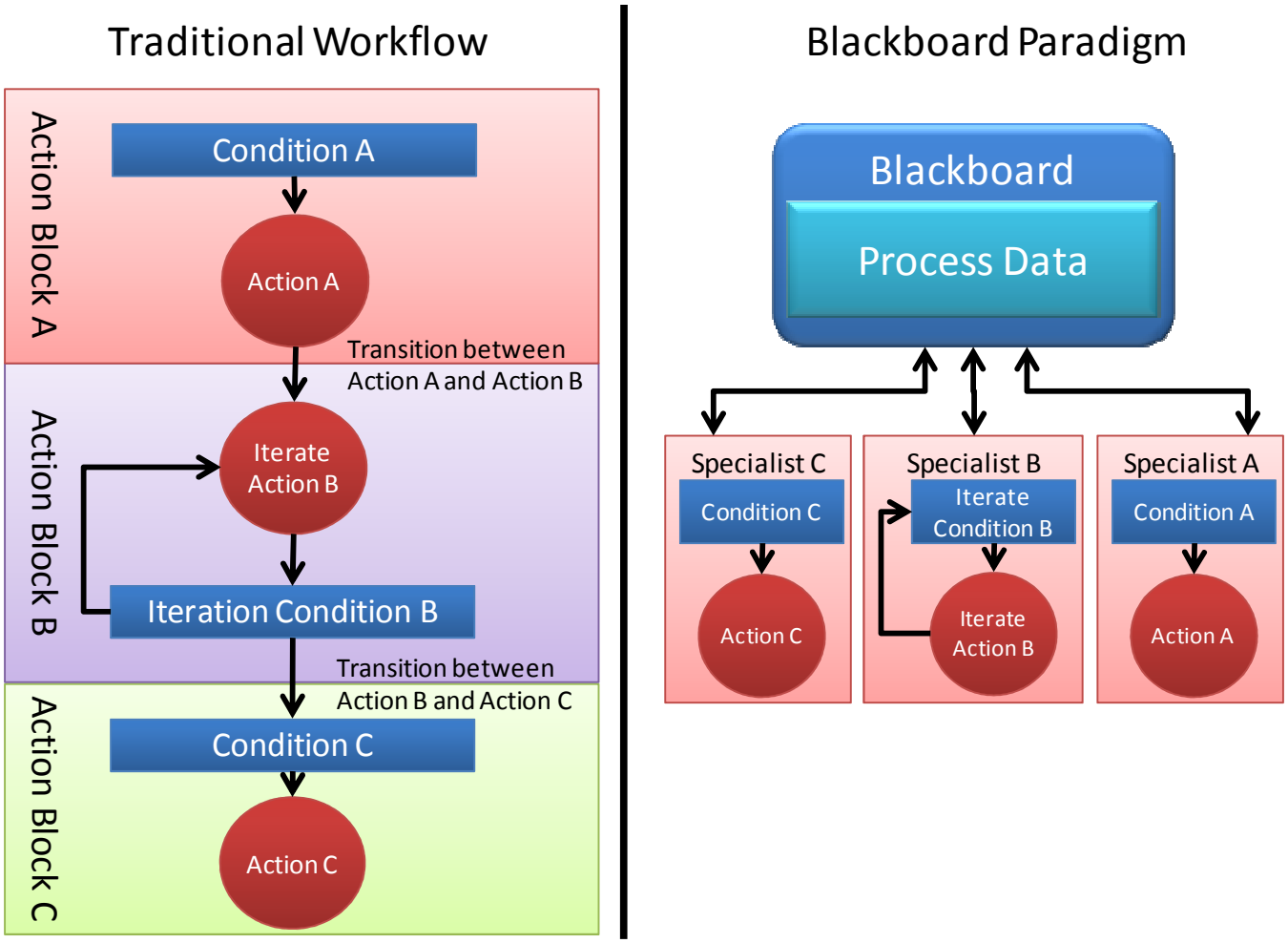

Figure 19: A comparison between the linear flow of a traditional workflow engine and the cyclic flow of an engine built on the blackboard paradigm.

From Figure 19, the hypothetical workflow process has to execute Action A, Action B and Action C sequentially. Additionally, Action B has to be repeated while Condition B is true (iteration workflow pattern). However, once Condition B is no longer valid, an action transition will take place and the workflow engine will evaluate and run Action Block C. It is important to note that once an action transition has taken place between Action B and C, most traditional workflow engines cannot regress back to the previous action block in this case the iteration workflow pattern executed in Action Block B.

With the blackboard paradigm, the specialist can execute its actions if the relevant workflow process data on the blackboard is available for them to do so. Moreover, specialists can execute their actions again if relevant workflow process data is available again. Some limitations of this can be spotted that may seem to render this paradigm impractical for orchestrating workflow processes. It can be seen that the locus of control for the blackboard paradigm depends on the data on the blackboard and the opportunistic response of specialists to changes on the blackboard. If there is no correct process data available for the specialists to apply their actions, the workflow

process might become frozen in a single state and never terminate. In the hypothetical workflow illustrated in Figure 19, Actions A, B and C have to be executed sequentially. If process data is available that allows Specialist C to execute Action C before Specialist A has had the chance to monitor the blackboard and respond to the process data, this workflow process might be incorrect. The blackboard architecture can potentially promote chaotic behaviour within a workflow process with actions being executed in an uncontrolled manner with specialists executing their actions whenever conditions or relevant data presents itself on the blackboard. These concerns just mentioned might be valid and this is discussed further in Chapters 5 and 6.

Opportunistic control in the blackboard paradigm might be useful in order to implement some of the complex workflow patterns as defined in [14] and [24] which are flexible enough to be altered by external stimuli during runtime. The interleaved parallel routing pattern (see Section 2.4 for more details) is one such example of how opportunistic control can be leveraged. Figure 9 shows an example of interleaved parallel routing where actions B, C and D can be executed in any order. In blackboard paradigm, three specialists can be created that are responsible for executing actions B, C and D. The three specialists will respond to three triggers placed in any timely order on the blackboard for each action to be executed. Eight out of twenty main workflow patterns defined by [14] and [24] can be identified where opportunistic can possibly be leveraged. This is explored further in Chapter 6.

One disadvantage of using the blackboard paradigm is the complexity involved in designing a workflow process. Workflow designers will require knowledge on how a blackboard system works in order to create workflows. The blackboard system can be seen to have its own domain specific workflow language which is script based. Condition-action rules have to be inserted into specialists which closely resemble script-based workflow languages (see Section 2.8.5 for more on script-based workflow languages) in order to define workflows. However, like many other WFMSs, a user-friendly UI can always be built that abstracts away all complexities where workflow designers can create workflows by creating simple flowcharts and the relevant specialists can automatically be created in the background. This is discussed further in Chapter 6.

Another difficulty that might exist in the blackboard system is the ability to execute concurrent threads within a workflow process. A workflow process can be split into multiple threads of control that are executed concurrently. The blackboard is a centralized data structure and only one entity can alter a piece of data on the blackboard at any one time. Multiple specialists cannot alter the same piece of data at the same time which might restrict concurrency within workflows and can prevent the implementation of some of the main workflow patterns. This discussion will proceed further in the next section.

3.5 Serial Vs Concurrent Blackboard Implementation

Recall from Section 3.2.6 that serial blackboards prevent multiple specialists from interacting with the blackboard at the same time while concurrent implementations allow some form of concurrency with specialists interacting

asynchronously, in parallel on one centralized data structure. In the context of developing a workflow engine, specialist concurrency will be required. Workflow engines should allow the execution of workflow processes where a single process thread splits into multiple process threads that are executed concurrently. Consider the following examples of an insurance claim that has to be registered which then fires off two parallel processes which check the customer's policy and assess actual damages. Figure 20 depicts this where the check customer policy and the access damage actions can happen concurrently. Within the blackboard system, a specialist will be created that is responsible for each single action (blocks within Figure 20). For this example, 5 specialists (5 action blocks) will be created. A serial implementation of the blackboard system will prevent the access damage and check customer policy actions to be executed at the same time. If the access damage specialist is engaged with the blackboard, the check customer policy specialist will have to wait until the blackboard becomes free. A serially controlled blackboard system will not be sufficient for implementing a workflow engine.

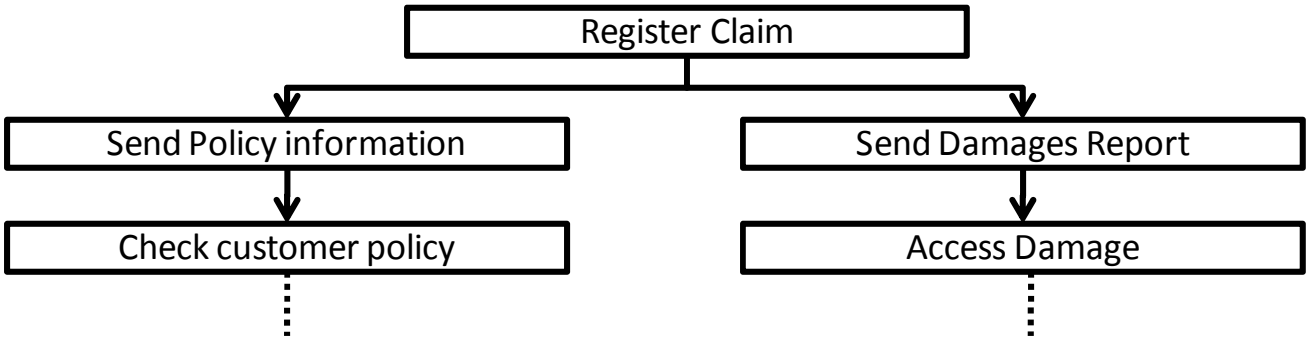

Figure 20: Parallel workflow process example.

A concurrent implementation of the blackboard system is essential for concurrent workflow process threads to be executed. To clarify, a concurrent implementation of a blackboard system should surpass the functionality, advantages and performance of serial blackboards by allowing concurrent execution of specialists, concurrent access of specialists to information to the blackboard and allow multiple problems to be solved at the same time (concurrently).

The SMBB approach as seen in Figure 15(c) allows each specialist to directly access the blackboard. This is the most direct approach to give concurrent access of the blackboard to the specialists. This approach can work well especially in a context of building a workflow engine with specialists engaging in many reads and writes at the same time.

For building a workflow engine, the SMBB approach can be taken. It can be argued that no matter what concurrent blackboard implementation is taken, there will always be memory contention on the blackboard and true concurrency can never be realized. In [79 - 81], it is suggested that specialists should be highly specialized (modular) to solve this problem. If this design suggestion is used, most memory contentions will be resolved. In the

context of building a workflow engine, a specialist can be designed by workflow designers for a single workflow task (to reduce memory contentions) and can also later be reused in other workflows.

3.6 Prior Research Concerning Workflow Engines and the Blackboard Paradigm

Research has been done regarding the feasibility of implementing a workflow engine using the blackboard paradigm [83]. This research did not use pure blackboard architecture. By adding a central controller (task manager) to control the specialists, the system presented in [83] has incorporated some aspects of layered architecture as well as retaining some aspects of blackboard architecture. The reasoning behind this was to curb potential chaotic behaviour as discussed in Section 3.4. Figure 21 displays the type of architecture employed in [83].

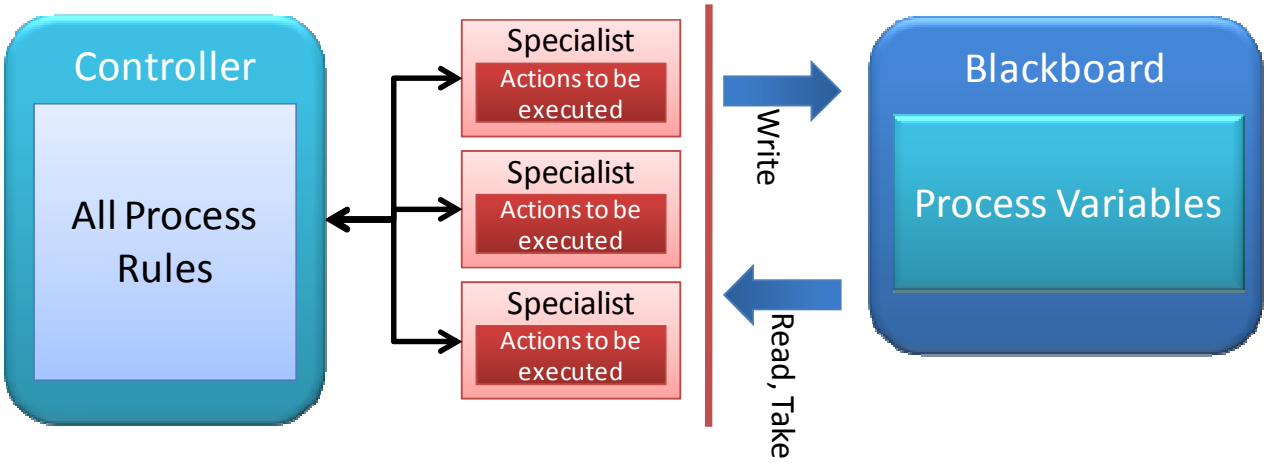

Figure 21: The architecture designed in [83] employs elements of layered as well as blackboard architecture.

From Figure 21, it can be seen that all the process rules are in the controller and specialists only contain actions. According to the blackboard architecture design guidelines defined in [16], specialists should have information about what conditions they can interact with on the blackboard. In [83], this information is housed in the controller which is uncharacteristic of the blackboard paradigm. According to [16], specialists should be autonomous and should not be controlled by the controller like passive drones. The system in [83] has the controller taking control of specialists by invoking specialist actions. Instead, specialists should be independent entities that contain rules that allow them to opportunistically engage with the content on the blackboard. The controller should be present just to coordinate access of specialists to the blackboard.

Blackboard systems are effective when there are numerous steps towards a solution and numerous potential paths involving those steps. By opportunistically exploring the paths that are most effective in solving the particular problem, a blackboard system can significantly outperform a problem solver that uses a predetermined approach to

generating a solution. By adding all process rules into the controller, E. Schikuta and K. Kofler have compromised on this opportunistic control which is one of the key aspects to the blackboard paradigm [83]. As already mentioned in Section 3.4, opportunistic control can be leveraged to implement complex workflow processes. From the evidence given, the architecture in [83] cannot be construed to be using the pure blackboard paradigm.

A prototype was built by S. K. Stegemann, B. Funk and T. Slotos in [84] using the blackboard architecture proposed by [83]. The support for basic workflow patterns (the first 5 out of 20 main workflow patterns) was used to evaluate the prototype. All the first five basic control flow patterns are supported by the prototype. The support for the complex workflow patterns were not investigated or evaluated by [84].

3.7 Summary

This chapter shows how a blackboard paradigm can be utilized to build a workflow engine. The way how a Petri-net is fired can be decomposed into a fundamental relationship that exists between three entities within a workflow process definition namely the workflow process rules, data and actions. These three entities can be mapped to components of the blackboard paradigm. The workflow process data can be placed on the blackboard and specialists that monitor the blackboard can contain process rules and the actions to be executed if the process rules are evaluated to true. Existing research that has been conducted in building a workflow engine using the blackboard paradigm is discussed. The research uses a quasi-blackboard paradigm to implement a workflow engine and the investigation into pattern support is limited and incomplete. The next chapter will attempt to provide some research objectives and contributions.

4 Research Question

4.1 Introduction

In Chapter 3, the components and behaviour of a workflow process was mapped to components of the blackboard paradigm. Researchers in this field of workflow management systems have defined five main aspects of workflow management along with their requirements and quality indicators, and have proposed this to be universally used to evaluate any workflow management system. These five main aspects of workflow management will be used to formulate the research question and objectives.

4.2 Research Scope

With perpetuating demands for larger and more complex workflow management systems, Workflow Management Systems (WFMSs) are being created by developers and researchers that offer different functionality and focus on some key areas such as flexibility, reliability, availability, scalability and interoperability just to name a few. These WFMSs are built in response to a very few set of requirements and ignore many other aspects of workflow management. The underlying cause of this is the absence of clear concordant guidelines or body of knowledge that can form some theoretical basis for workflow management just like relational algebra provides a theoretical platform for Database Management Systems (DBMSs) [85]. Standardization bodies such as the Workflow Management Coalition (WfMC) are attempting to establish standards in some areas such as workflow interoperability (most WFMSs however do not use this interoperability standard placed by the WfMC) but no consensus has been reached with numerous other aspects of workflow conceptualization and management [86].

In this unclear and complex setting, it becomes difficult to evaluate any workflow management solution as there are so many aspects and quality factors to consider. Some researchers have tried to address this situation. First and foremost, Van der Aalst in his seminal paper [14] has evaluated WFMSs in the context of workflow patterns. Investigating how many workflow patterns a WFMS can support is a sound, universally applicable technique that can be applied to any workflow engine since it does not require internal implementation details and architecture of workflow engines. Joblonski and Bussler in [87] define important aspects to comprehensive workflow

management. Petkev, Oren and Haller in their paper "Aspects of Workflow Management" have incorporated and refined many of these approaches including those proposed in [14] and [87] and have provided five key workflow aspects namely the functional, behavioural, informational, organizational and operational aspects to workflow management [88]. They also provide quality indicators for each workflow aspect that can be applied to verify their support in WFMSs [88]. These five aspects of workflow management can be used to assess a workflow engine built on the blackboard paradigm by applying quality indicators from each aspect.

This approach of evaluating a WFMS built on the blackboard paradigm using these five perspectives of workflow management has been chosen since it is universally applicable to any WFMS and it can be used to compare WFMSs to one another. The following sections consider each aspect in detail as described in [88].

4.3 Aspects of Workflow Management and Quality Indicators

There are five main aspects to workflow management namely the functional, behavioural, informational, organizational and operational aspects [88]. These aspects are described in more detail below as they are presented in [88]. Besides these, additional aspects for evaluating WFMSs also exist which are security, causality, integrity and failure, history and quality. This research will focus on the five main aspects although the remainder of the aspects will be touched upon in passing.

4.3.1 Behavioural Aspect

The behavioural aspect defines when and in what order activities and tasks are to be executed. The 'control flow' or 'process model' defines the ordering of activities and event handling. According to van der Aalst in [14], the behavioural aspect is the most important while all other aspects are secondary. All other aspects are built on the 'control flow' or 'process model' and hence rely upon it. This argument is further supported in that other aspects of WF management are outside the control of a WFMS. For instance, the production data (informational aspect) and execution of actions are handled by external resources such as other applications and humans (organizational aspect). The WFMS only initiates actions or tasks depending on the 'control flow' and cannot control the execution of these tasks or actions. The 'control flow' or 'process model' embodies the main functionality of the WFMS; thus it is logical that this aspect should be given prominence.

This aspect can be modelled by a plethora of formalisms and modelling techniques ranging from textual descriptions [89], semi-formal activity diagrams [90] to formal Petri-net [91] workflow models (already spoken about in Section 2.3). None of these techniques has gained ascendency and achieved ubiquitous usage [92]. These modelling tools are useful to define workflows and provide analysis techniques for improving efficiency and check for correctness in workflow processes. Workflow process designers can use a particular modelling tool or technique in conjunction with a workflow engine to create correct and efficient workflows.

Van der Aalst has created an approach through creating 20 main workflow patterns that can be used to evaluate the behavioural aspect of WFMSs regardless of any modelling tool or technique which has become the benchmark for evaluating the behaviour of workflow engines. Workflow patterns can be directly used to evaluate the behavioural aspect of any workflow engine while being impervious to its implementation details. A workflow pattern "is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts [93]." These workflow patterns determine the level of workflow behavioural functionality in a comprehensive manner and provide a way to compare WFMSs. Van der Aalst compared 15 leading workflow engines in 2003 and found that 51.6% of 20 main workflow patterns could be implemented. The same study was repeated in 2006 with new versions of some of the same workflow engines and some new ones (14 workflow engines in total) and found that 58.4% could be implemented [24]. Besides the 20 main workflow patterns mentioned in [14], a further 22 more workflow patterns are also defined in [25] that are essentially extensions and special cases to the main 20 workflow patterns. Indeed many more workflow patterns can be indentified but special focus should be given to the 20 main workflow patterns before proceeding further beyond. This research will focus on the 20 main workflow patterns.

For evaluation purposes, a workflow engine built on the blackboard paradigm should be subjected to these workflow pattern tests. Hypothetical workflow process definitions can be created within the blackboard paradigm driven workflow engine to simulate the 20 workflow patterns. A workflow pattern is simply a flow of control between tasks. Therefore, the execution order of activities has to be observed within a running workflow process instance to verify that a workflow pattern can be implemented on a workflow engine built on the blackboard paradigm.

4.3.2 Informational Aspect

The informational aspect defines how data flows and what data is consumed and produced. The aim of the informational aspect is to provide the correct data at the right time so an activity in a workflow process can be successfully executed. Data inside a workflow can be dissected into control data which is used to orchestrate workflows and production data which is used and produced by a business process. Data and data flow can be modelled by Entity Relationship (ER) diagrams [94], object role models [95] and class diagrams. Techniques that model the informational aspect have gained limited use [14]. The majority of WFMSs focus upon the workflow process definitions (control flow) coupled with their modelling techniques.

'Data patterns' have been introduced in [15] to the workflow management arena by Russell, ter Hofstede, and Edmond in order evaluate the level WFMSs can represent, transport and utilize data within a workflow. Just as workflow patterns capture types of control flow, data patterns capture types of data flow. 40 main data patterns have identified in [15] that were used to evaluate six leading WFMSs. It was found that 48.3% of the data patterns could be implemented by these WFMSs.

These same data patterns listed in [15] are used to evaluate a workflow engine built on the blackboard paradigm. Each data pattern has to be demonstrated within a hypothetical workflow process scenario.

4.3.3 Other Aspects

The functional aspect defines what the workflow does. This aspect characterizes different kinds of workflows and the constraints placed on these workflows. There are two types of workflows namely 'simple workflows' which are devoid of sub-workflows and 'composite workflows' which consist of one or many sub-workflows. Workflows need workflow definitions that are subject to three constraints. The 'enter constraints' are applied when a workflow is spawned. The 'exit constraints' is to check if the workflow has ended properly. The 'runtime constraints' are to check for consistency between all transitions of the workflow.

The quality indicators provided by [88] for the functional aspects of workflow management are as follows:

- Workflow engines should allow for flexibility when creating workflow definitions. This will allow for the creation of ad-hoc workflows where the rules of the workflow can be slightly altered in response to ever-changing requirements.

- The creation of sub-workflows within a workflow should be allowed. A workflow definition should be reusable in the creation of other workflows. Additionally, dependencies between sub-workflows should be able to be defined.

- While defining a workflow, dependencies and constraints should be allowed to be inserted. These dependencies and constraints will be applied before a new workflow process is spawned to ensure that sufficient information and resources are provided to the workflow process for it to successfully be executed.

The objective of this research is to build a functional workflow engine using the blackboard paradigm that satisfies the quality indicators mentioned above. It is imperative to realize that this research revolves around how well the blackboard paradigm can accommodate these quality indicators and not how the quality indicators could be somehow moulded or forced into the blackboard paradigm. This research is all about remaining true to the blackboard paradigm. The blackboard paradigm cannot be used and would be unviable for building a workflow engine if any one of these quality indicators cannot be satisfied (due to the nature of the blackboard paradigm) or their implementation into the framework of the blackboard paradigm is too difficult, unfeasible or unsatisfactory.

The operational aspect defines how a workflow activity can be implemented and executed within a workflow. Workflow activities are implemented in the form of application programs or scripts that run these applications (outside the workflow engine). The workflow engine has to interact with applications which can be completely dissimilar in nature and it has to invoke various functions offered by these applications. Application Plug-ins have

to be created within the WFMS that invoke these various application programs and expose an Application Program Interface (API) that the core workflow engine can invoke. According to [88], Application Plug-ins interfaces are required to have the following characteristics made available for the core workflow engine to call:

- The parameters for an action to be executed by an application program have to be provided. The interface should be able to accept parameters for executing actions.

- To improve efficiency, information should be provided to determine whether an application program should be run synchronously or asynchronously. An invocation mode setting should accompany each application program action.

- An option should be provided which indicates whether user interaction is required. An interaction mode setting has to be offered.

- This interface should implement control coupling to ensure that the workflow engine is passing application plug-ins information on what actions to perform.

- Parameters that are passed to an application program plug-in that is run multiple times might change depending on the business process. Therefore the system should allow passing of ad-hoc parameters with values that can be only determined at runtime and static parameters that can be hard-coded. Application program settings should also be allowed to be defined.

Conversely, the application programs plug-ins should be required to supply the workflow engine through this interface with the following information:

- If an application requires interaction from a user; once the user selects and performs an action, this should be relayed by the workflow engine.

- Status and error codes should be provided by the application programs to inform the workflow engine regarding the successful execution of a particular action.

Currently, there are no quality indicators available that can used for the comparison and evaluation of WFMS APIs. However, a set of interaction patterns will be introduced in the near future that attempts to mimic the approaches in [14] and [15]. So in order to fulfil this aspect, a workflow engine built on the blackboard paradigm should at least be required to have these characteristics in its application plug-ins' interfaces that couples it with the application programs.

The organizational aspect defines who will engage with or perform a task and through what tool or interface. The agent that performs that task could be human or non-human. The organizational aspect also defines a hierarchy of agents, their roles, security and access authorizations and group affiliations. All of this is done outside the workflow

engine in the realms of the invoked applications, administration and monitoring tools and workflow client applications. This research is primarily focused on the workflow engine component of the WFMS and how or if other components of the WFMS can be able to fit into the workflow engine. Therefore, this aspect of workflow management will be ignored.

4.4 Research Question

The above four aspects of workflow management (the organizational aspect is excluded) can be used to state the research question. The workflow engine built on the blackboard paradigm must at least be functional and operational and meet the quality indicators put forth by functional and operation aspects of workflow management. These two aspects of workflow management have to be fulfilled completely before the research can proceed any further. The main quality indicators for assessment are the workflow (behavioural aspect) and data (informational aspect) patterns that can be used to compare and evaluate different WFMSs. It is important to note that these patterns are not comprehensive but form a commonly occurring subset. These patterns provide a common way for assessing the capabilities of workflow engines. To fulfil the requirements set out by the behavioural and informational aspects of workflow management, the blackboard-based workflow engine should produce a desirable result of maximising workflow and data patterns support and comparatively outperforming most leading commercially available workflow engines in the level of workflow and data pattern support. In the light of these four aspects of workflow management, the research question is:

> *"Can the blackboard paradigm be used to implement a functional and operational workflow engine that can fulfil the requirements set out by the behavioural and informational aspects of workflow management?"*

4.5 Research Contributions, Methodology and Validation

In order to verify the research question, a prototype workflow engine is developed. This prototype is implemented using the blackboard paradigm. The guidelines provided by [16] for blackboard systems are strictly followed.

The prototype system can potentially be used by the University of Witwatersrand's Computer and Network Services (CNS) division. It therefore needs to be integrated into the CMS framework currently running the Wits CMS called Silent Bob. Another requirement is that there should be a workflow designer (a web UI; a workflow client application) that can allow users to create, edit and remove workflow processes. This prototype workflow engine built using the blackboard paradigm is affectionately named "Wits Enterprise Workflow Engine" (WEWE).

The development of WEWE consists of three stages. The first development stage of WEWE aims to produce a workflow engine. The second stage should produce the workflow API (WAPI) to lay the foundation for process

definition tools, invoked applications, administration and monitoring tools and workflow client applications to be attached. Additionally, some of the functionality of the process definition tools and invoked applications should be developed. The final stage aims to deliver a full Workflow Management System (WFMS) depicted in Figure 11. This research primary delves into the first stage of development; only the workflow engine.

To answer the research question, test instances of all the 20 main workflow patterns documented in [14] and [24] will be created and executed inside WEWE. The correct order in which workflow activities are executed as defined within each workflow pattern needs to be observed for each test instance. The number of successfully executed test cases for each workflow pattern will evaluate WEWE. One concern highlighted is that a workflow engine built on the blackboard paradigm might display chaotic workflow behaviour as described in Chapter 3. During testing, the workflow processes have to be observed for chaotic behaviour. For some test cases, one can gauge whether or not individual workflow patterns are supported. However, further test cases are developed that combine individual workflow pattern test cases together and this is where test cases might fail. The manner in which these test cases are created is discussed further in Chapter 6. Similarly, tests have to be conducted to examine the support for the 40 main data patterns. Support for some data patterns can be verified through inspection. Workflow pattern test cases can be reused to inspect the variety of different ways in which workflow process data flows that are captured by these data patterns. For specific data patterns, additional WF test cases have to be developed that capture the essence of these data patterns. These WF test cases are run inside WEWE and the support for these data patterns have to be verified by inspection. More details regarding how these data pattern tests cases are developed are provided in Chapter 6.

4.6 Summary

This chapter has presented the research question. The research objective is to determine how well a blackboard-based workflow engine can function. The workflow engine is evaluated using the quality indicators of the four main aspects of workflow management as defined by [88]. The next chapter presents the design and implementation details of WEWE.

5 The Prototype - Wits Enterprise Workflow Engine (WEWE)

5.1 Introduction

This chapter presents the design and implementation details of the prototype. WEWE was designed and developed to verify if the blackboard paradigm can be used to implement a workflow engine. To simplify the design process, it was split into four stages. In the first three stages the three main components of the blackboard paradigm were designed starting with the blackboard, specialists and ending with the orchestrator/controller. The last stage involved the design of the ancillary components (WF client apps, WF agents, logger, communication layer etc) which fall outside the blackboard paradigm but are necessary to render the workflow engine application operational. Each blackboard system component was designed in accordance with the guidelines and stipulations laid out in [16].

5.2 The Blackboard

The blackboard is a global data-structure. Every component inside the blackboard system (orchestrator/controller and specialists) needs to interact with the blackboard (global data-structure). The design of the blackboard component is critical since it affects all other components within the system. Careful consideration had to be given as to how the specialists would efficiently monitor (performance) and interact with the blackboard. The blackboard also needed to be structured to allow workflows to be easily executed.

5.2.1 Overall Structure

Generally, workflow engines allow the creation of a workflow definition which is the description of some business process with enough information (tasks and control flow description) to be able to be executed by a WFMS. An executing instance of this workflow definition is called a ‘workflow process instance’ or a ‘case’ [20]. At any one time there can be multiple workflow process instances concurrently running but which are completely independent from each other. For instance, consider the example in Section 2.7 of a workflow for an employee submitting a

leave application. In a workflow engine, this workflow for employee leave applications will be saved in the workflow entity that houses all the workflows within the workflow engine. Any leave applications that are received will create a new instance of the workflow that will be encapsulated as a workflow process instance. It is clear that there exists a one-to-many relationship between a workflow and its associated instances. This fundamental relationship in WFMSs, shown in Figure 22, formed the basis on which the blackboard's structure was designed.

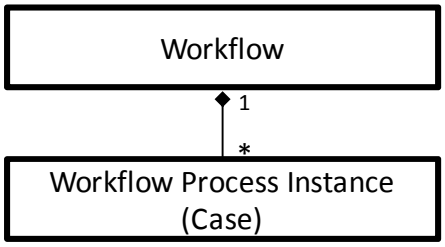

Figure 22 : A UML diagram of the fundamental relationship between workflows and process instances.

It is stipulated in [16] that the blackboard should have objects that are hierarchically organized. These objects can be organized into zones on the blackboard with which specialists can interact. The relationship in Figure 22 assisted in organizing the blackboard hierarchically. Firstly, a 'workflow process instance' object was created containing all process and control data of a single process instance. Functionality that can be used by specialists and the controller/orchestrator for retrieving information, editing, deleting and adding data was also added to this object. A 'workflow' object was created to contain all the workflow process instance objects. To state it plainly, the 'workflow' object is used for grouping workflow process instances of the same type of workflow. These workflow objects containing their WF process instances can be placed on the blackboard. Thus, a simple and intuitive object hierarchy of two levels was used to structure the blackboard shown in Figure 23.

It can be seen from Figure 23 that the blackboard is split into zones for each workflow process instance. This is beneficial as a set of specialists for a specific workflow process instance can specifically monitor information pertinent to that WF process instance instead of arbitrarily scanning large irrelevant portions of the blackboard to find information of relevance. The staging zone is completely separate from the WF and WF process instance zones. It is used by the orchestrator/controller to prepare new workflow processes. Once the new WF processes are setup, the orchestrator transfers these new workflow process instances into their relevant WF zones.

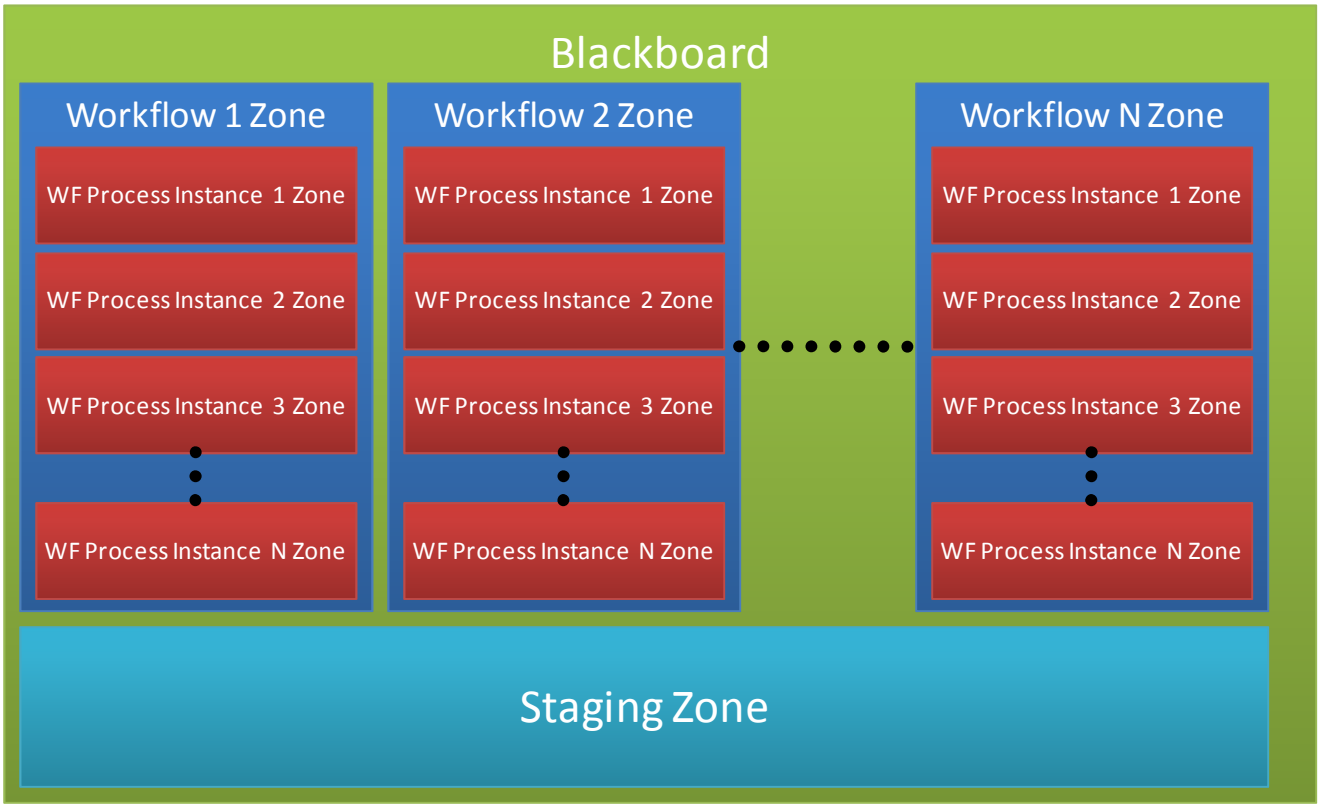

Figure 23: The conceptual structure of WEWE's blackboard.

5.2.2 Implementation

Implementation details of how the blackboard component is built are contained in Appendix A. It is recommended that the reader read through this section to gain a better appreciation and understanding of this component. Remember that a shared memory blackboard (SMBB) approach is taken to build the workflow engine. With the SMBB approach, multiple specialists are allowed to access the blackboard at the same time. In the context of a single workflow process instance, multiple specialists should be able to gain direct access to the workflow process instance zone. A locking strategy is employed that prevents many specialists writing to a single piece of WF process data within the WF process instance zone at the same time while allowing multiple specialists to read any piece of data concurrently.

A blackboard of this nature using this locking strategy was easily implementable using Java's map data structures specifically the concurrent hash map data structure [96]. The blackboard was designed to essentially consist of a concurrent hash map of workflow objects and these workflow objects further consist of a concurrent hash map of WF process instances as shown in Figure 24. Specialists that want to monitor and interact with a particular workflow process instance (a zone) can hash the blackboard by simply using the ID of the workflow and the ID of the WF process instance to retrieve the required WF process instance object. If a new WF process is spawned, the

orchestrator/controller will create it inside the staging zone and prepare the object before transferring to its WF zone. Implementation details regarding this are provided in appendix A for the avid reader.

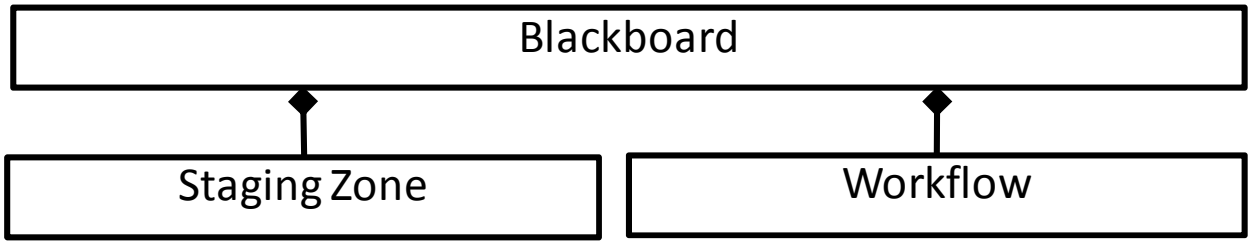

Figure 24: UML diagram of the blackboard structure.

All the workflow information for each workflow process instance is stored in a memory table in the workflow instance zone as elements or entries within the concurrent hash map. The inherent concurrent capabilities of this native Java data structure provided the necessary locking strategy. For multiple specialists (which execute in separate threads) trying to retrieve or add a new entry, no locks are engaged for that map entry. When a specialist (thread) is modifying an existing entry, a lock is engaged only on that entry forcing other specialists (threads) to wait for that entry to become free. It is important to note that locks do not occur on the map level but rather on an individual entry level [96]. This enables many specialists to read and write to many entries at the same time.

Typically, the workflow information is split into production data (data used and produced by the business process) and control data (used by the workflow engine to execute a workflow process instance). These two types of data can be altered; however it is important that some information fields should remain static. For instance, a workflow can be created which requires the date of birth (DOB) of a client. In this example, the DOB cannot change under any circumstance and no workflow action or task should be allowed to change this information. However, specialists in typical blackboard systems are unrestricted in changing any type of information on the blackboard and this might lead to possible data corruption of business processes. To curb this problem, the traditional arrangement of workflow information as either production or control data was discarded and more general data abstraction of splitting data into variables or constants was chosen. WF information defined as *variables* can be altered or removed by specialists and, by extension, workflow tasks and activities. WF information defined as *constants* cannot be changed throughout the lifetime of a workflow process instance. Specialists and, by extension, activities or tasks cannot alter or remove constants once they are inserted into the blackboard. Specialists can however read and use constants and variables for their use. The workflow process instance object is structured as depicted in Figure 25.

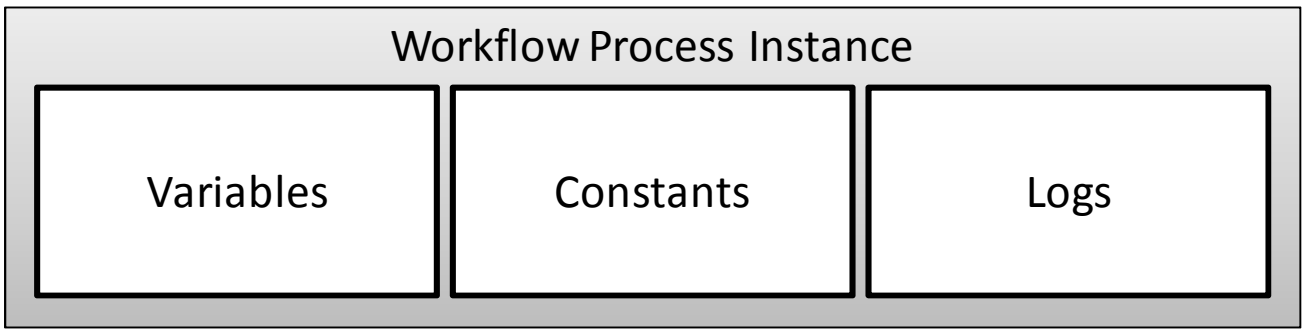

Figure 25: Structure of a workflow process instance.

As the workflow process instance is running, specialists modify the data inside the workflow process instance as to determine the path of the process instance and to execute tasks and actions. For auditing purposes, all changes to the workflow process instance are recorded in the logs component of the workflow process instance.

The blackboard component in blackboard systems remains completely in memory so specialists can quickly monitor and interact with the blackboard. In the context of workflow management, it is dangerous not having a persistence mechanism as information on the blackboard for a long-running workflow processes might be erased due to a system reload or crash. Therefore, a database exists to save the data changes to workflow process instances. In the event of a system reload, the workflow engine is able to revive unfinished workflow processes and they can resume from the point of stoppage.

The storage database has been designed to enable full auditing similar to the database design implemented in date tracking system for oracle databases [97]. Appendix A.2 discusses the database design structure.

Both constants and variables consist of three attributes listed as follows:

- The *name* which can be used as a key within the concurrent hash map to efficiently retrieve the constant or variable
- The *value* of the constant or variable
- The *type* which assists in the serialization, persistence to the database and de-serialization

More details about how the variable or constant types are used and the different kinds of data types that were implemented can be found in Appendix A.3.

WF process instances are required to store each variable's and constant's name, value and type. Specialists should be quickly able to extract a variable's or constant's value or type by supplying the WF process instance object with the name. A data structure was required to store key-value-type 3-tuple. A concurrent hash map within a concurrent

hash map was used. Code Listing 2 shows that format of how variables or constants are stored within the concurrent hash maps.

```
 [
[DOB]:                          [[value]: ”May 10, 1987”, [type]: String]
[Age]:                          [[value]: 25, [type]: Integer]
[ApplicationForm]:      [[value]: ”/docs/App1234.pdf”, [type]: document]
[ApplicationAccepted]:          [[value]: false, [type]: Boolean]
]
```

Listing 2: Format of constants and variables stored in WF process instance objects.

5.3 The Specialists

A large degree of design choices are present when creating specialists. Depending on the application, the design of specialists could vary from a single condition-action pair (used in [68]) to assert statements to a forward-chaining rule-based system to linear-programming (used in [76]) to a complex neural network approach (used in [81]). In the context of this work, it does not matter what type of design approach specialists are made to follow as long as specialists can monitor and make a meaningful contribution to the blackboard. However, a design approach should be taken that is appropriate for the application. In the context of executing workflows, the process rules within a workflow (see Figure 17 for the relationship between process rules and other components of a workflow process) are integral to defining the flow of execution of activities within a workflow. Specialists should be designed to accommodate the creation of these process rules. The principals and design of rule based systems can be used to create process rules that can successfully execute workflows. This is discussed further in the next section.

5.3.1 Rule Based Systems

The concepts behind forward-chaining rule-based systems were used to design the specialists for WEWE. Figure 26 shows the main components of any rule based system.

The fundamental component of any specialist is the condition-action pair. Coincidently, the fundamental building block of Rule Based Systems (RBSs) is also the condition-action pair [98]. A group of these condition-action pairs are used to make a RBS. As seen in Figure 26, these condition-action pairs form the knowledge base of any RBS. The objectives of RBSs are to make decisions or to solve a problem by emulating decision making through triggering these condition-action pairs against known facts housed within the workable information. This is done by the inference engine of any RBS system where logical reasoning processes are constructed. Depending how these condition-action pairs are arranged and triggered, the inference engine of RBSs can either use the forward-chaining or backward-chaining to achieve its objectives [99]. Backward-chaining is goal-driven. This technique starts from a set of goals and by using the workable information available, tries to prove or achieve these goals. This technique is not possible in the context of a workflow execution since all workable information regarding the goals (such as

order of execution of actions) is not provided beforehand and depends upon external stimuli. However, forward chaining is a good fit. It is a data-driven process that uses the data available to compare against its rules in order to determine which actions to fire off. Additionally, actions sometimes involve the addition, alteration or deletion of data within the workable information. Evaluating rules from the workable information (WF process data), executing actions (WF process actions) and in turn altering the workable information is the essence of workflow execution of activities as shown in Figure 17 (Section 3.3).

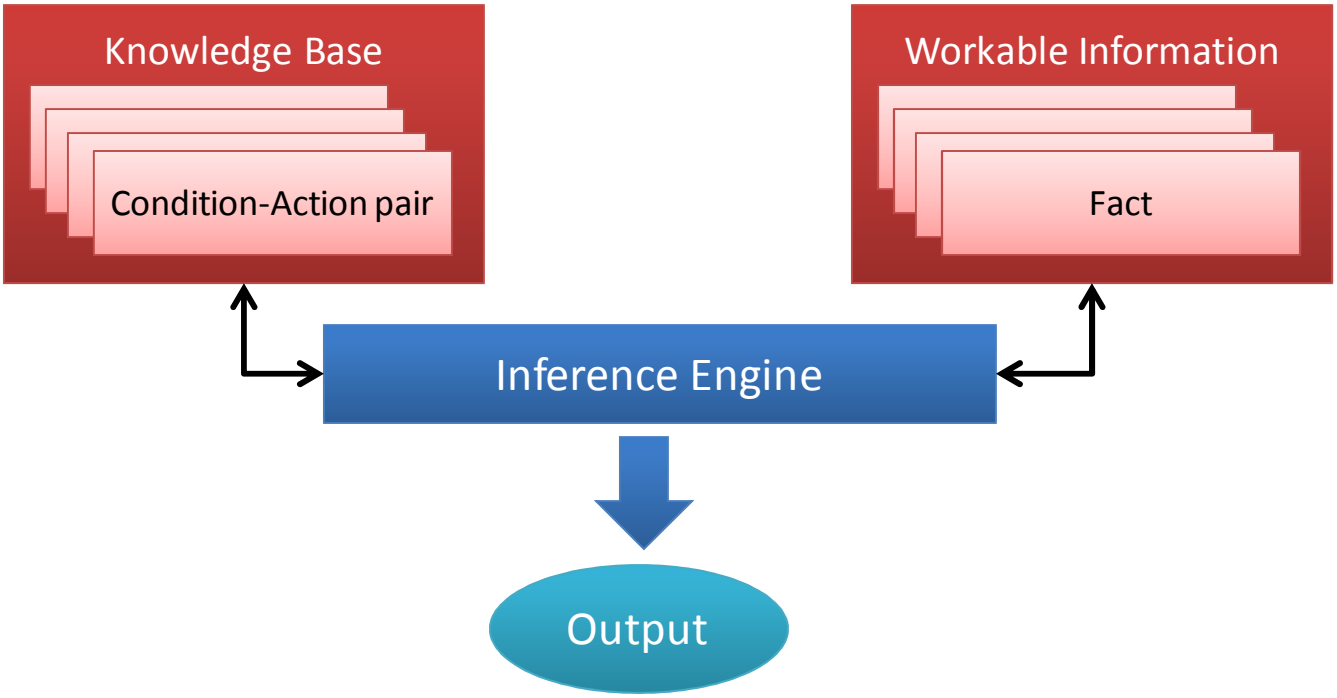

Figure 26: Components of a Rule Based System.

5.3.2 Specialist Rules Implementation

RBSs contain 'rules' or 'attributes' which are essentially condition-action pairs. The condition is typically a list of if statements (the antecedents) and the action is a set of functions to execute (the consequents). The condition can be made up of many antecedents that can be joined by the 'AND' and 'OR' keywords. Typically in RBSs, the 'AND' keyword (conjunction) is used to conjoin antecedents together and the 'OR' keyword (disjunction) is used to separate antecedents. This convention is used to simplify implementation and to prevent ambiguity when evaluating conditions with multiple antecedents. An example is provided of an antecedent-consequent pair in code listing 3.

```
IF (<antecedent 1> AND <antecedent 2>) OR (<antecedent 4> AND <antecedent 5>)
<consequent>
```

Listing 3: Example of an antecedent-consequent pair.

However, the placement of brackets between antecedent clauses can change the order in which the clauses are executed. An example is provided in code listing 4 to demonstrate this. The first antecedent-consequence pair has antecedents conjoined together with the 'OR' keyword and disjoined by the 'AND' keyword while in the second pair the same antecedents are separated by the 'OR' keyword and joined together by the 'AND' keyword.

```
IF (<antecedent 1> OR <antecedent 2>) AND (<antecedent 4> OR <antecedent 5>)
<consequent>

IS NOT EQUIVALENT TO

IF (<antecedent 1>) OR (<antecedent 2> AND <antecedent 4>) OR (<antecedent 5>)
<consequent>
```

Listing 4: Example to demonstrate the effect of bracket placement on antecedent evaluation.

The defining of bracket placement between antecedent clauses can be accomplished for specialist rules utilizing different techniques. One technique is to specify whether the keywords ('AND' and 'OR') are antecedent conjunctions or disjunctions. Another technique is to break up the antecedent clauses and push them into a queue that will be evaluated in the order based on the bracket arrangement. Ultimately, a technique using nested 'IF' statements was formulated that does not break the traditional RBS conventions (using 'AND's and 'OR's to join and separate antecedents respectively). The nested 'IF' statement can be used to fabricate an antecedent evaluation result identical to the result if the keyword conventions are reversed (using 'AND's and 'OR's to separate and join antecedents respectively). This is illustrated in a few examples listed in code listing 5.

```
IF (<antecedent 1> OR <antecedent 2>) AND (<antecedent 4> OR <antecedent 5)
<consequent>

IS EQUIVALENT TO

IF ((<antecedent 1>) OR (<antecedent 2>))
       IF ((<antecedent 4>) OR (<antecedent 5>))
              <consequent>
```

```
IF (<antecedent 1>) AND (<antecedent 3> OR <antecedent 4>) AND
(<antecedent 2>)
<consequent>

IS EQUIVALENT TO
```

```
IF (<antecedent 1>)
        IF (<antecedent 3>) OR (<antecedent 4>)
                IF (<antecedent 2>)
                        <consequent>
```

Listing 5: Examples of using nested IF statements to produce the same results as antecedent-consequent pairs expressed with "AND" and "OR" conjunctions.

The antecedent of a rule comprises of the object linked with its value using an operator. The operator essentially identifies the object and assigns the value.

In RBSs, the object is retrieved from the workable information. In the blackboard paradigm, the object is retrieved from the blackboard. The blackboard information is split into either variables or constants. It is important for the object to distinguish whether the object should be retrieved from the 'constants' or 'variables' component of the workflow process instance object on the blackboard. Therefore, the specialist rules will be written as shown in code listing 6.

```
If (const.name == "John Doe" AND const.StudentNumber == 00000A AND
var.applicationApproved == TRUE)
        <approve John Doe's application>
```

Listing 6: The format of specialist rules is provided using an example.

From code block 6, the 'var' and 'const' can be seen as data specifiers for variables and constants. The constants cannot change during the lifetime of the workflow process while variables can be altered by specialists. This can be explained through a business perspective where a hypothetical application is submitted to a workflow that has a specialist with a rule shown in code listing 6. The application has the applicant's name and student number which can never change within the lifetime of the workflow processes and therefore are saved as constants on the blackboard. However, the application can either be approved or declined during the workflow process and this piece of information is saved in a variable.

To accommodate time-based triggering in workflow executions; the 'timelapsed' specifier has been incorporated. The time lapsed specifier provides the amount of time that has passed since the creation of the workflow process. Unlike the data specifiers ('const' and 'var') that allow data to be filtered on the blackboard into constants and variables, the time lapsed specifier has to provide the time lapsed in either seconds, minutes, hours or days. An example of how this can be utilised in a specialist rule is shown in code listing 7.

Incorporating time-based information and calculations into the blackboard system were particularly difficult as it can possibly break the blackboard paradigm if one is not careful. From [16], it is stipulated that specialists should be static repositories of knowledge and time-based calculations have to be performed without altering specialists. The blackboard is a repository of information that can only be accessed and changed by the specialists or the

orchestrator/controller. Initially, a global specialist was designed to continually update the time lapsed information for all WF process instance objects. Testing revealed this technique to be inefficient as the increase of WF process instance objects degraded performance. The updates to the time lapsed information were too slow and sporadic. This potentially can lead to incorrect and outdated time lapse information being sent to the specialists. Instead of continually updating the time lapse information by external entities (specialists and orchestrators/controllers), time lapsed information should be produced whenever specialists require them. The current timestamp is constructed by the specialist. The time lapsed is calculated by subtracting the creation timestamp (found on the blackboard in the workflow process instance object) with the current timestamp (from the specialist).

```
IF (timelapsed.days > 30 AND var.bookoverdue == true)
       <place overdue fine for reader>
```

Listing 7: The format of specialist rules making use of the 'timelapsed' specifier.

In addition to rules for time lapsed in workflow executions, timeout scenarios have to also be accommodated. If a specialist is required to perform a specific set of actions within a certain amount of time but is unable to, then the specialist rules cannot be satisfied. A timeout feature performs a set of actions when a timeout occurs. The timeout feature for specialists can be construed as another kind of specialist rule (condition-action pair) which dictates the actions to be performed when a timeout occurs. Timeout rules are written in the form shown in code listing 8.

On the surface it might seem that there is no noticeable difference between timeouts and time lapsed. The only distinction between the two is that timeouts will only be true if the specialist has never contributed to the blackboard (successfully executed one of its rules). Time lapsed rules will always be applicable whether or not the specialist rules have been successfully executed.

```
IF (timeout.days >= 7)
       <execute timeout actions>

IF (timeout.hours >= 2  AND  var.actionACompleted == false)
       <perform Action A>
```

Listing 8: The format of timeout specialist rules.

The timeout information for each specialist is recorded within the orchestrator/controller when specialists are loaded into the blackboard system. Once a timeout occurs for a particular specialist, the orchestrator/controller notifies the specialist that its timeout has occurred. The specialist will run all the attributes and execute actions (if rules are satisfied) that include timeout rules. Therefore centralised time-based event triggering of specialists has been used to implement the timeout feature in WEWE. This prevents individual specialists from constantly polling for timeouts.

5.3.3 Inactive and Active Specialists

In the context of workflow management, workflow process actions only occur when there is a change to the workflow process variables. Furthermore, the duration between any two actions occurring in a business process could be hours or even days. It is wasteful and fruitless for specialists to be actively monitoring the blackboard looking for changes to the workflow process variables and trying to execute workflow process actions. It is prudent that some form of event based triggering of specialists should be implemented. Specialists should only engage with the blackboard when there is a change to the blackboard. Two forms of specialist triggering were implemented as follows:

- Specialists belonging to one workflow process instance will be triggered (become active and start monitoring the blackboard) when any changes are made to the workflow process instance object.

- Specialists that have logged their timeout information into the orchestrator/controller will be triggered (notified to become active and to run their timeout attributes) when their timeout occurs.

The internal workings of specialist event-based triggering will be discussed once the orchestrator/controller component of the blackboard system is explained (in Section 5.5.3).

There is an interesting distinction between specialists being inactive (dormant) and becoming active. From [16], it is stipulated that specialists should be static repositories of knowledge. However, once a specialist becomes active, the information inside the specialist can possibly change while the specialist is making a contribution to the blackboard. For instance, in WEWE the specialists create a new current timestamp as a data member to determine the time that has lapsed during the workflow processes. The insertion of a new data member (the timestamp) can be construed to be a specialist change although it does not alter the basic specialist definition. A more critical example would be specialists altering their rules or actions while being active. Changes to specialist definitions are not allowed in blackboard systems according to the guidelines presented in [16].

In order to prevent changes to the specialist definitions and content, most conventional blackboard systems ([67], [68] and [76]) create specialist instances of the specialist definitions. A specialist instance is essentially a copy of the actual specialist but the information inside the instance can be changed. The specialist instance can be animated in order to execute its rules (condition-action pairs) and try to contribute to the blackboard. Once the specialist instance has finished executing its rules, it is destroyed.

In WEWE, a specialist animator exists with each inactive specialist definition. It takes the specialist from its inactive state and animates it. It can be seen as a container in which the specialist instance can be dropped into to be animated. Specialist animation involves the linear execution of all specialist rules (condition-action pairs) neglecting any rules with timeout conditions. If a timeout occurs, the specialist animation involved only executes the timeout rules.

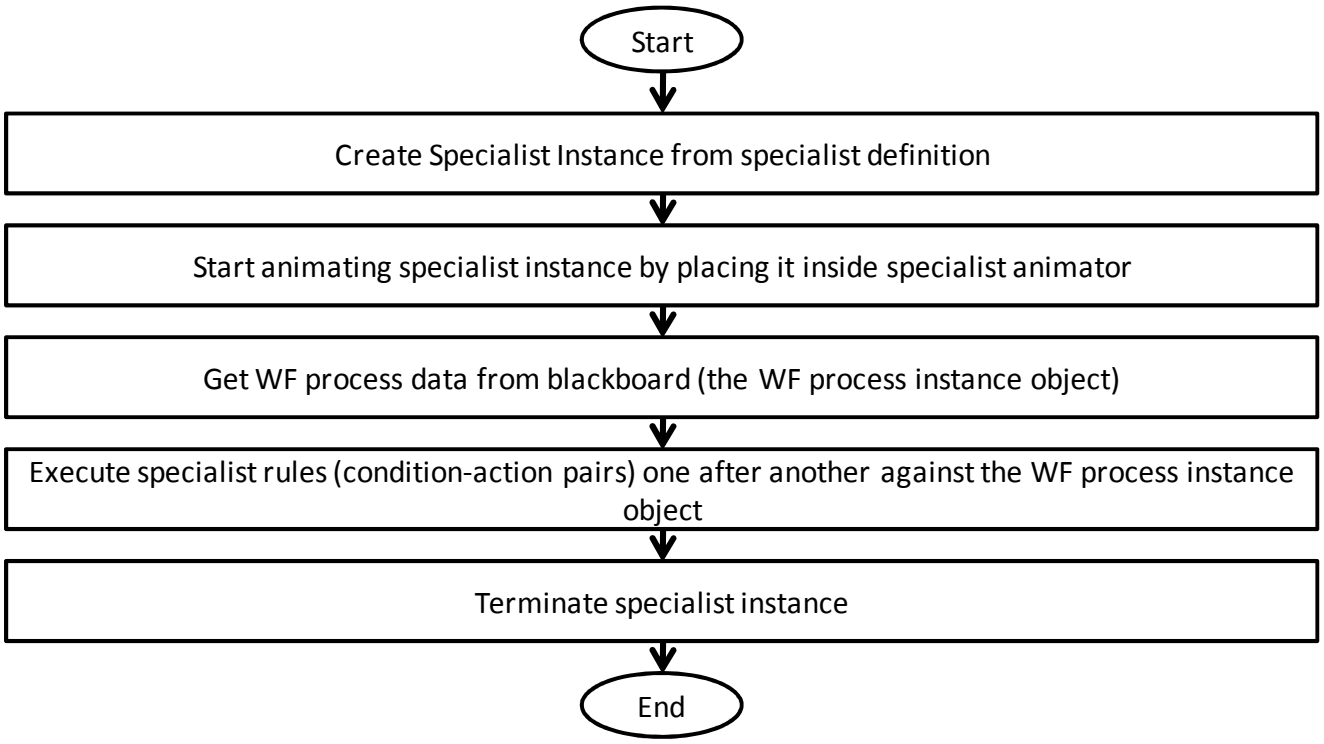

Figure 27: The process undertaken by specialists when they become active.

The linear execution of specialist attributes was implemented to reduce complexity. This implies the importance of the order of specialist rules as the first rule is most significant and the last rule is the least significant. In the context of building a workflow engine, a linear rule execution strategy is useful to execute workflow activities in order. Traditional RBSs employ a more rigorous approach and trigger rules that carry a likely possibility of being executed successfully based on the workable information [98] which is not necessary in WEWE as the linear rule execution strategy can suffice for executing workflow activities. Specialists follow the process illustrated in Figure 27 in order to interact with the blackboard and ensuring that the contents of specialists remain unchanged.

5.3.4 Specialist Creation and Persistence

The workflow process definition tool for WEWE is an internet application with a thin client that allows users to create specialists and their rules through a JavaScript imbued interactive interface. With very little assistance from the backend (server), the client interface uses JavaScript objects (more specifically JavaScript object literals) to temporarily store the information inserted by the user regarding all specialists and their rules. Once the user is satisfied with all the rules of a particular specialist and inserts it into the definition of the workflow, all the specialist rules stored in the JavaScript objects are serialized into a compact JSON string. This JSON string consisting of the specialist attributes accompanied with the specialist's metadata (name and description) is sent to the backend (server) to be persisted. Implementation details regarding the creation and persistence of specialists are provided in Appendix B.2.

5.3.5 Specialist Actions Implementation

Once a specialist rule set has been evaluated to true, a set of actions have to be performed. The design of how actions are executed within specialists is of particular importance since it directly affects the operational aspect of WFMSs as defined by [88]. WEWE can execute workflow activities by invoking internal application plug-ins within the WFMS. These application plug-ins can be built to either execute workflow activities without any outside assistance or can be built to hand over execution of workflow activities to external applications. WEWE can invoke these application plug-ins through a common interface. In summary the operational aspect requires the WEWE interface with application plug-ins to accommodate:

- The definition of parameters that are to be passed to these application plug-ins.
- The setting of an interaction mode which indicates whether user interaction is required.
- The setting of the invocation mode that indicates whether an application task action should run synchronously or asynchronously.

Furthermore, this interface should implement control coupling to ensure that WEWE is invoking the application plug-in and passes it information on what actions to perform. Details regarding the implementation of how application plug-ins are invoked is provided in Appendix B.2.

Along with the details regarding the actions to be performed, an interaction mode setting can be set as well to define whether or not user interaction is required. This setting is for application program actions that require user interaction that can produce two distinct results depending on the interaction mode. For instance, an application can contain an action which provides the user with a form to approve an order. If the integration mode is set to ‘auto’ (instead of user), this application action might behave differently, by for example, automatically approving the order based on a predefined set of rules. For applications that do not require user interaction, the interaction setting is set to ‘auto’.

For efficiency, the workflow engine should be able to run actions asynchronously or synchronously. Workflow designers decide which actions should be run synchronously, and which asynchronously. Users of the workflow should be oblivious to this distinction despite it being of importance to WF designers. A set of synchronous actions run consecutively and may require the results from preceding actions in order to run. Asynchronous actions run in their own thread and can be made to run independently from the executions of other actions [99]. Taking heed of this distinction, actions in WEWE can be split up and placed into action sets. New threads will be created for each action set to execute the actions consecutively within the action set. If one action needs to be executed independently, this action can be placed into an action set with no other actions. If action B requires the result from action A, actions A and B should be placed within the same action set. More details regarding how specialist actions are executed are provided in Appendix B.2.

5.4 Blackboard Agents

Before the orchestrator/controller (see Section 5.5) is explained, another integral component of WEWE, as a fully fledged WFMS, needs to be discussed, namely the blackboard agents. Apart from the specialists, other components of the WFMS (the process definition tools, invoked applications, administration and monitoring tools, and workflow client applications) outside the main engine (see Figure 11 in Section 2.6) will also need to post new, or alter existing, information on the blackboard. For instance, the workflow client applications component of the WFMS offers user interfaces for users to initiate workflow actions and essentially control the flow of any workflow process. A user might initiate an action or make a workflow decision and the relevant information for this action has to be posted into the correct zone (the workflow process object) of the blackboard. Specialists monitoring this blackboard zone will instantly engage with this new data in an attempt to execute their rules (condition-action pairs) and by extension perform workflow process actions leading to information on the blackboard being added or altered. Specialists will in turn engage with this new data on the blackboard hence continuing this opportunistic cycle of control.

Allowing external applications surrounding the main workflow engine to post information to the blackboard is an important design consideration. The components surrounding the main engine can be on different machines that can have completely dissimilar software environments. Furthermore, the fact that software systems from outside the WFMS are able to post information onto the blackboard raises security concerns. In this highly heterogeneous environment, interoperability between the main engine and other systems becomes a critical issue.

An entity needs to exist which will be able to operate in a networked environment and interact with and transport information between different software components and systems. In WEWE, many components expose integration interfaces using different communication protocols and techniques, for example, Remote Method Invocation (RMI), JSON-Remote Procedure Calls (JSON-RPC), Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) web services. This entity has to facilitate the exchange of information between the main engine and other systems through these integration interfaces in order to post changes to the blackboard.

Software agents can perform the role of this described entity. Software agents are entities that perform predefined tasks that are on some level autonomous, capable of learning and cooperating together to achieve a common goal. A consensus has not yet been reached of what classifies as an agent since this term is in vogue, with any entity that exhibits some sort of learning, cooperation or being autonomous being attached with this loose label [100]. In spite of this, agents can be loosely defined as “computer programs that simulate a human relationship by doing something that another person could do for you” [101]. Figure 28 paints a typical scenario in WEWE of a user initiating a hypothetical workflow action, ‘action A’. An entity exists which receives the relevant information pertaining to ‘action A’, performs ‘action A’ (posting data on the blackboard to initiate action A) on behalf of the

user before reporting back to the user. This entity can be defined as an agent because it fits the definition of an agent provided.

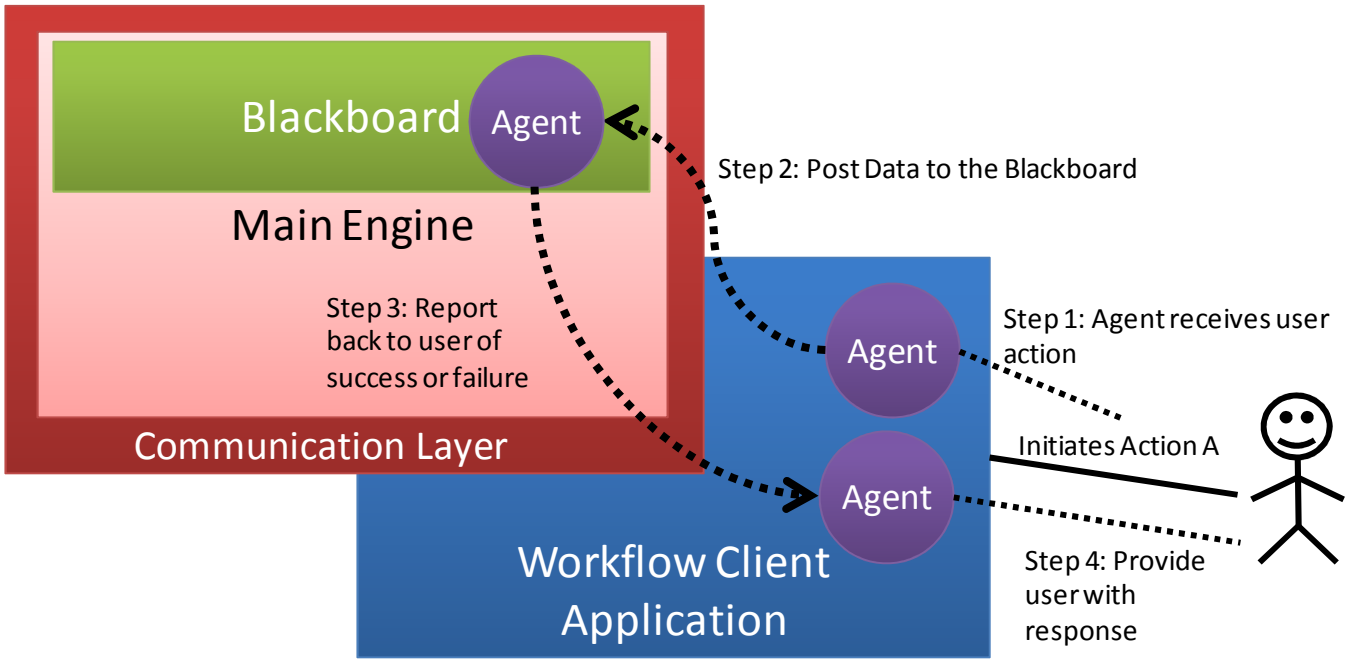

Figure 28: A typical scenario of a user interacting with WEWE's workflow client application.

Different types of agents have various degrees of autonomy, learning and collaborative ability depending on the goals and requirements of the application in which they are deployed. The closest type of agent that best fulfils the requirements of WEWE is mobile agents. Mobile agents are software components that can transfer information in interconnected systems and perform a set of objectives on behalf of the user and report back to the user.

5.4.1 Agent Implementation

There are many components within the WEWE WFMS that need to interact and exchange data with each other and the main workflow engine. Some components expose a REST or SOAP web service for communication while others allow remote method invocation on Java methods. In WEWE, agents are built to encompass knowledge regarding which communication protocol to invoke in order to transfer data. The agents also consist of the CRUD operation to perform on the desired data repository. The agents themselves are serialized and are passed from one component to another in this serialized state and are de-serialized when they have reached their component destination as seen in Figure 29. Once inside the component destination, the agents can update the data repository (such as the blackboard) and report back to the initiator.

Figure 29: Agent transport process between two components within a WFMS.

5.4.2 Types of Agents Created

There are four main kinds of agents that have to be created which are create, read update and delete (CRUD) agents. These agents do not only apply CRUD operations to the blackboard but also to other data structures or repositories within the WEWE environment. Additionally, there is a user agent that communicates with the human users or external systems.

Lastly, a WF process spawn agent is created which allows users to create new WF processes. The WF process spawn agent is an extension of the Create agent. Instead of creating only one WF variable or constant on the blackboard, the WF process spawn agent packs all data, objects, variables and constants required to start a new WF process instance into a compact collection and travels to the blackboard to create a new blackboard zone on the blackboard. Information on exactly how a new WF process is created is discussed in the next section.

When a workflow action is initiated either through a human user using workflow client applications (applications residing inside the WEWE WFMS that interact with human users) or an external system, various data structures and repositories usually have to be updated in the WEWE environment as well. This is done through a *transaction* which comprises of many updates to different data structures including the blackboard. For instance, a workflow can exist which represents the business process of promoting an employee. The Human Resources (HR) manager will initiate a workflow to promote a specific employee. This starts a transaction that will perform the following tasks:

- Add new data to the workflow process information on the blackboard to advance the workflow process.
- Update the employee's details to reflect this promotion on the Lightweight Directory Access Protocol (LDAP) directory which might reside as a client application within the WEWE framework.
- Update the Oracle Human Resources Management System (HRMS) which is a system outside of WEWE.

This transaction needs to be atomic and if one of the update queries fail, the entire transaction needs to be rolled back. The user agent will receive the workflow action from the user which resides in the workflow client actions domain. The user agent is implemented as a static agent to simplify transaction handling. The user agent will then delegate all queries that it cannot perform to other agents.

The user agent will delegate adding information to the blackboard to a Create mobile agent which will post workflow data to the blackboard and then report back to the user agent.

A designated Update mobile agent will update the LDAP directory for the user agent. The LDAP directory does not actually physically dwell within the WEWE framework. A REST API to LDAP is available as a client application in WEWE. The Update mobile agent will use this REST API to execute the HTTP PUT requests to update relevant information on LDAP and then return back to the user agent.

The last Update will post a message to execute the update query on the Oracle HRMS (external system) and then report back to the user agent. The Oracle HRMS uses messaging queues to allow for integration and asynchronous communication with other systems. The last Update mobile agent will connect to the Java Message Service (JMS) server specifically WebSphere MQ [102] (used at Wits University and integrated to be part of the WEWE WFMS) using STOMP (Simple Text Oriented Message Protocol) and drop the update query in the relevant queue where it will be processed later by the HRMS. If all delegate agents perform their task successfully, the user agent will report back to the user (HR manager) of success. Otherwise, the user agent will have to rollback the transaction and report back to the user of failure.

From this example, agents can be seen to be cooperating with each other but are not autonomous. Cooperation is achieved by user agents delegating tasks to other agents while maintaining the session (period in which the transaction is alive) and monitoring the transaction. Franklin and Graesser in [103] explore and propose a formal definition of an autonomous agent which “is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda.” The agents in WEWE do not have their own agenda. They have a single drive which is to act as external users and systems tell them to. Their singular agenda is to consult and be commanded by the user or an external system.

All components external to the main engine are developed with the Java Spring Framework [104] that helps provide some basic scaffolding for the components to speed up development. The transaction and session management within agents are handled by Spring’s transaction and session libraries. Integration between components using REST and SOAP Web-Services, RMI, RPC, etc which agents use to transfer themselves and their information between different components are all implemented using Spring.

Security was an obvious concern in the development of agents for WEWE. While the system needs to enforce security policies, it must not be completely rigid as to hamper integration and interoperability between components.

Spring's security library [105] was used to secure WEWE. With Spring security, a user, external system or any entity can be authenticated using authentication providers to establish its identity. The owner of this identity (also known as the principal) is assigned different roles. So when an external party assumes control of an agent to do their bidding, the external party needs to go through the authentication process and all the party's credentials and roles are stored within this agent. With spring security, every component of WEWE is secured and given certain user roles. Only agents with these user roles can access the component. When an agent wants to transfer information to a certain component, it will pass Spring security's proxy layer to be authorized based on the roles required. The security of WEWE can be encompassed within the organizational aspect of workflow management. Although this aspect is important in its own right, it is not being investigated in this research as discussed in Section 4.3.3.

5.4.3 Comparison with Specialists

Agents can be seen to be very similar to specialists. Both consist of rules and certain actions to perform. However, there are many subtle differences that exist between them.

For specialists, complex and open ended customized rules can be defined by workflow process designers. For agents, the rules are rigidly predefined depending on the type of agent.

Specialist actions could consist of performing CRUD queries against the blackboard and also the performing of workflow activities by initiating workflow application programs. Agent actions only perform CRUD queries against the blackboard or any other type of data source.

Specialists can only monitor and update a certain zone of the blackboard. Agents have the ability to perform CRUD queries on any data source including the blackboard within the WFMS.

Specialists cannot communicate with each other as stipulated by [16]. Agents have the capability of cooperation between other agents to achieve a common objective.

An agent's lifetime is much more short-lived than a specialist's lifetime. An agent exists until it performs some sort of query against a data source and reports the results back to the initiator. A specialist's lifetime is directly tied to the lifetime of the WF process instance it is monitoring.

5.4.4 Agents and Interoperability of WFMSs

Researchers that are prominent in this field commonly cite poor interoperability and to lesser extent incompatibility with external systems as major limitations in WFMSs [7, 8, 15 and 94]. With the introduction of agents that are able to transfer data in and out of WEWE, these limitations are diminished. External systems can pass data into and request data out of WEWE and WEWE can request data and pass data to external systems by using agents.

5.5 Orchestrator/Controller

A significant challenge encountered in the design and development of most blackboard systems is in selecting and implementing an optimal opportunistic control strategy. It is important to realize that the blackboard is data driven with the triggering of appropriate specialists when new data is inserted onto the blackboard [106].

The controller/orchestrator can be visualized to be a global specialist since it needs to monitor and act upon the entire blackboard and not just a specific zone like a conventional specialist. There are several responsibilities for the orchestrator which are listed as follows:

- Start-up, management and termination of all workflow processes
- Handling sub-workflows within composite workflows
- Specialists triggering based on changes to the blackboard or timeout events
- WF process tracking and analytics
- Handling of ad-hoc workflows

5.5.1 WF Process Management

One of main responsibilities of the orchestrator is to fulfil some of the quality indicators of the functional aspect of WF management. This is specifically in the handling of spawning and termination of new and completed WF processes respectively.

Designing a new workflow by the WF designer entails the creation of rules that control the flow of defined actions, constraints and the definition of any initial data that is required to spawn a new workflow process. The defining of the required initial data (termed dependencies and constraints for a WF) is an important quality indicator for the functional aspect of WF management. These dependencies and constraints need to be checked before a new WF process is spawned to ensure that sufficient information and resources are provided to the WF process for it to be executed successfully.

The orchestrator is responsible for spawning new WF processes and checking dependencies. To start a new WF process, the entire initial data for that WF process is placed into the staging zone of the blackboard. To take an example, a WF process can be spawned when filling out and submitting an online form on the web. The data from the form is packed inside a WF process spawn agent whose sole objective is to deliver this data to the blackboard's staging zone. The orchestrator will check all the dependencies and constraints defined for the WF against the initial data for newly created WF process. If all the checks pass, the orchestrator will create a new WF process object

(new blackboard zone) within the relevant WF zone and move all the initial data from the staging zone to the newly created empty WF process object.

The orchestrator creates a brand new WF control thread for every new WF process. This WF control thread loads all the specialists required for that WF process. Specialist monitoring, triggering and animating are all handled by this WF control thread. The orchestrator keeps track of this WF control thread until the WF process is terminated. When a WF process is terminated, the orchestrator will then simply end the WF process control thread thereby indirectly terminating all the associated specialists, and remove the WF process object from the WF zone.

5.5.2 WF Analytics and Management

The orchestrator keeps track of all the WF process control threads. Entities within the WFMS such as administration tools can query information regarding all the WF processes. For instance, an administrator can query for all the WF processes within a certain WF that have reached some point in the workflow.

The orchestrator also exhibits specialist behaviour by having the ability to change any part of the blackboard. This is useful for administrators who want to administer workflows. Situations may arise when workflow processes are stuck in an error state. Workflow process errors will require the attention of a system administrator. The administrator will then attempt to resolve the issue and then resume the workflow process again. The system administrator is able to use the orchestrator to change the workflow state in order to resume the WF process.

Additionally, circumstances may arise of loosely defined ad-hoc workflows that may have changed during runtime. The orchestrator can be used to either skip or alter the flow of workflow activities by adding or altering WF states to skip or reorder those workflow activities respectively. Section 5.5.7 contains more information regarding ad-hoc workflows and how the orchestrator is utilized to manage this kind of workflow.

5.5.3 Concurrent Specialist Triggering

It has been established that some form of specialist triggering is required (in Sections 3.2.4 and 5.3.3). Specialists do not constantly monitor but are triggered based on the following events:

- A change in the workflow process instance made by either an agent, specialist or the orchestrator
- A specialist timeout

The producer-consumer pattern is used to implement specialist triggering since it is ideal for handling multiple concurrent processes (specialists running in different threads) and buffered communication between multiple processes [107].

The WF process control thread consists of specialists for that WF process. Every specialist consists of a specialist animator which involves the running of all specialist rules (condition-action pairs) against the data inside the WF

process object. The specialist animator's function is to continually animate the specialist making the specialist constantly monitor and try to interact with the blackboard. The WF process control thread allows all of its specialists (independent threads) to run. Thus, specialists are concurrently engaged with the blackboard and the blackboard itself manages data consistency. However, it is useless to constantly monitor the blackboard if none of the specialist's rules are successfully executed as already established. So a condition exists in the specialist animator to pause when no specialist rules are executed successfully. Simply put, specialists will be animated and continually be animated until animation is fruitless and thereafter the specialist is paused.

Each WF process control thread consists of a linked blocking queue which is used to resume or trigger these paused specialists. Any changes to the WF process object made by a specialist, agent or orchestrator or if a specialist timeout occurs (specialist timeout is discussed further in Section 5.5.4) all will result in a token being put into the Linked Blocking Queue (LBQ). If there is a token on the LBQ, any paused specialists will be resumed by the WF process control thread. Figure 30 depicts how the producer-consumer pattern was implemented for specialist triggering.

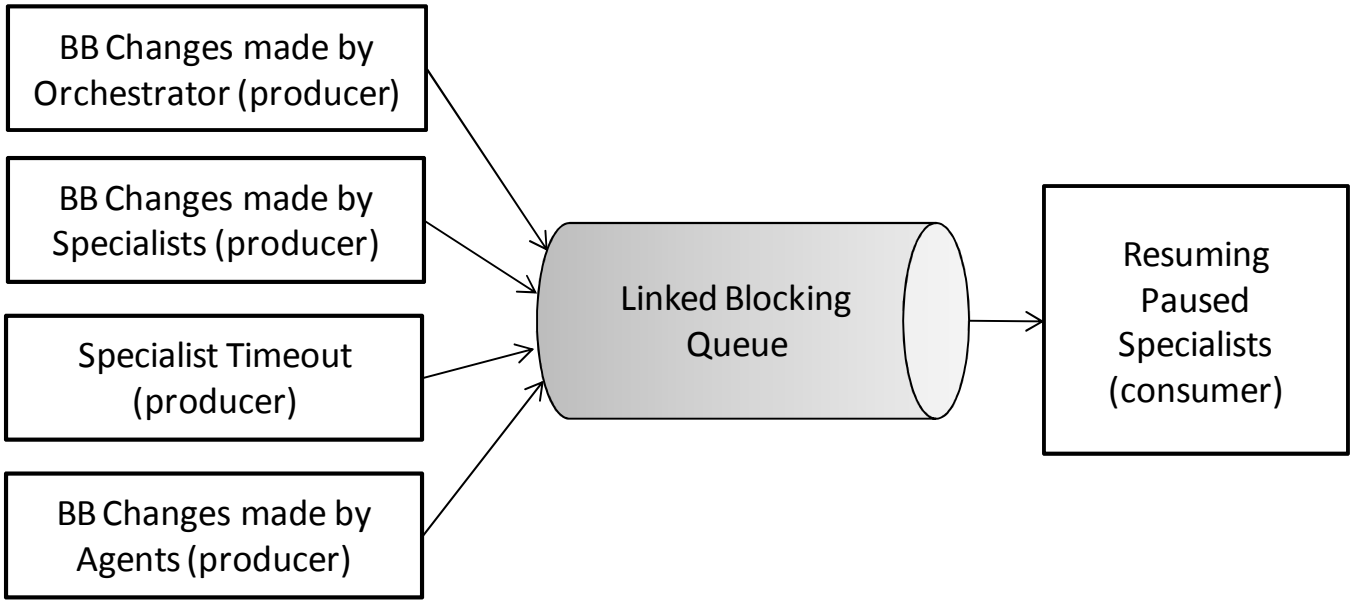

Figure 30: Producer-consumer pattern was used for specialist triggering.

This use of a LBQ is beneficial as there is no need to constantly check if there are any tokens placed on the LBQ. The LBQ simply blocks the WF process control thread (pauses thread at that point where a token is retrieved) until a new token is placed. Therefore, the WF process control thread only becomes active when there is a change on the blackboard resulting in all specialists being triggered; otherwise it will remain paused signifying a paused WF process [108].

A situation may arise where a token is retrieved and all the paused specialists resume animation. However, there might be specialists that are already active which are midway in their animation and might miss a change to

contribute to the blackboard again. A token placed on the LBQ means another round for every single specialist to animate their rules. Therefore, the WF process control thread will only remove a token from the LBQ after all specialists have been animated once for that token.

It is important to mention that by implementing this triggering technique the specialists are not communicating with each other as stipulated by [16]. Any specialist that has made a change to the blackboard simply puts a token on the LBQ. To follow a more strict definition of a blackboard system, this LBQ can be placed on the blackboard. Despite it not being implemented this way for WEWE, it is suggested for the future that the LBQ be moved to the WF process object (zone) on the blackboard.

When a specialist or orchestrator alters any part of the blackboard WF process zone, it must place a token in the LBQ for that WF process control thread. For specialist timeouts and agent CRUD queries, specialist triggering might be a little different as discussed in the next two sections.

5.5.4 Specialist Timeout Triggering

All specialists that are created log their timeout information into the orchestrator. The timeout information is saved using a hash map with each specialist's unique ID. Using the JAVA's native countdown latch library, a countdown is immediately started for each timeout [109]. As soon as the countdown reaches zero for any specialist, the orchestrator will place a token on the LBQ located inside the WF process control thread to which that specialist belongs to.

Orchestrators can be seen to behave as specialists, but this timeout triggering component of the orchestrator that allows the contents of the orchestrator to change breaks the definition of a specialist. Specialists are static entities. Orchestrators deviate from this definition since their contents are constantly changing.

5.5.5 Agent Execution Triggering

The mediator design pattern was used to enable agents to place tokens on the LBQ [110]. An agent interface component (the mediator) was implemented inside the orchestrator. This design prevents agents from accessing the blackboard directly. By using the orchestrator's agent interface, the agents can use it as a interface to the blackboard shown in Figure 31. The agents will access the agent interface which in turn will conduct actual updates on the blackboard.

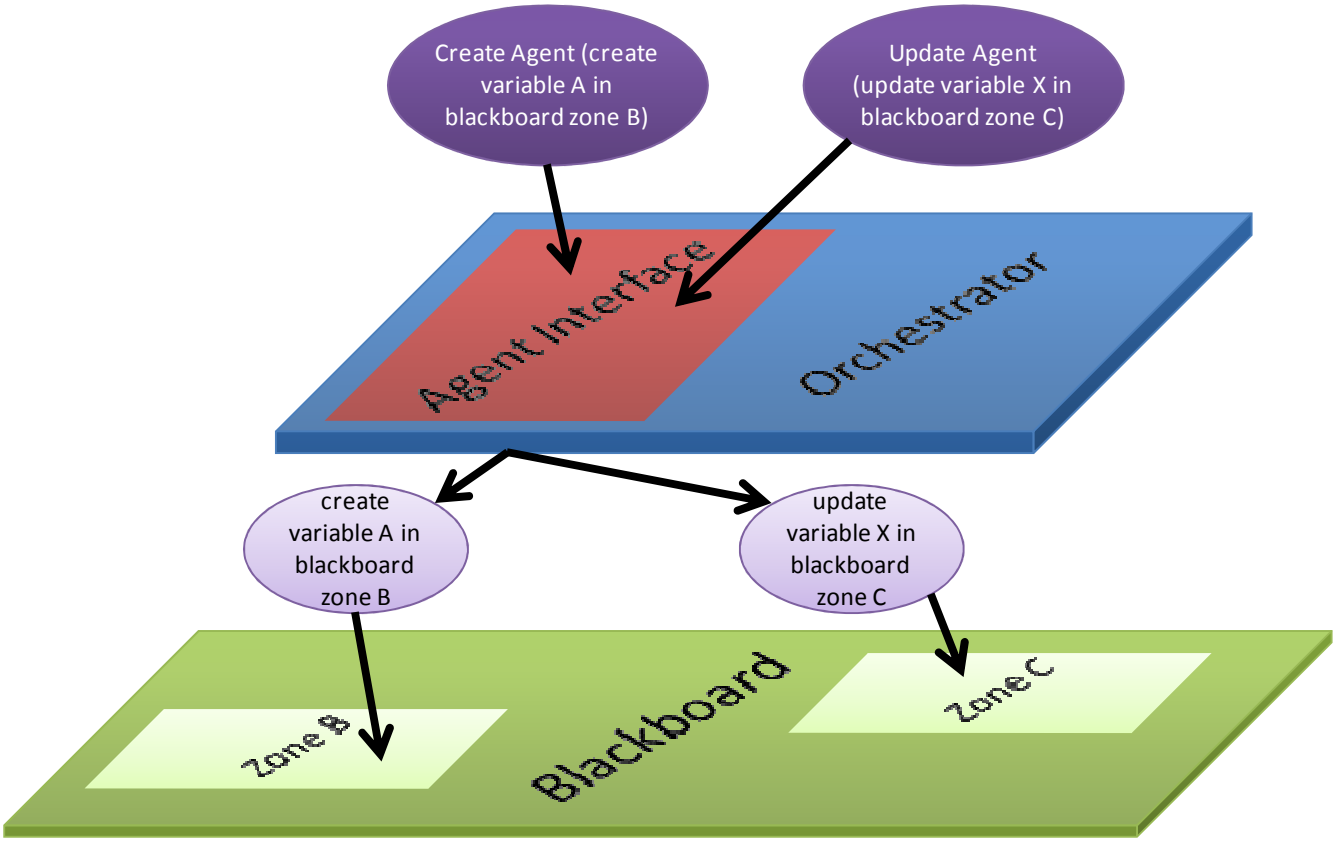

Figure 31: The way in which the mediator pattern is used to handle agents updating the blackboard.

The agent interface has knowledge of which blackboard zones (workflow process instance objects) have been updated and put tokens on the LBQs belonging to these WF process instance objects.

5.5.6 Sub-Workflow and Composite Workflows

There are two types of workflows namely *simple workflows* which are devoid of sub-workflows and *composite workflows* which consist of one or many sub-workflows. As part of the quality indicators of the functional aspect of WFMSs, the creation of sub-workflows within a workflow should be allowed. A workflow definition should be reusable in the creation of other workflows. Additionally, dependencies between sub-workflows should be allowed to be defined.

A sub-workflow is a normal workflow that can be run independently if required. However, it becomes a sub-workflow when it is spawned within a composite workflow (parent WF process).

The process of spawning a new normal workflow or a sub-workflow is very similar. The only difference is the origins of the initial WF data to start the WF process. A normal WF process will receive this data from outside the WF engine. For a sub-workflow, this initial WF data comes from the WF data currently existing within the parent WF (inside a WF process object or zone). An application plug-in program (for specialist actions) was implemented that allows the WF designer to define what data from the parent WF is used for the spawning of the sub-workflow.

This initial data is copied to the staging zone on the blackboard where the normal process of spawning a new WF process will occur.

Once a sub-workflow has ended, it needs to move the WF process data required by the parent WF back to the blackboard zone of the parent. Once again, the decision is made by the WF designer when the workflow is created.

It can be seen that the creation and termination of sub-workflows is essentially an exercise in moving WF process data from one blackboard zone (WF process objects) to another. A component in the orchestrator exists that efficiently moves WF data from the staging zone to WF process object (zone) during the spawning of new workflows. It is convenient to reuse this functionality in the handling of sub-workflows. The application plug-in programs that are implemented use the component of the orchestrator that can move data from one blackboard zone to the next.

5.5.7 Ad-hoc Workflows

Workflow engines should have the ability for WF designers to create flexible workflows namely ad-hoc workflows. The WF designers should be allowed to change the rules of a workflow even during its execution in response to ever-changing requirements. A small web-interface was built for workflow management as mentioned in Section 5.5.2. Functionality was added to the orchestrator and this web-interface to allow reloading of different specialists inside an already active workflow process. If the rules of a particular workflow process instance are altered due to a change in requirements, new specialists can be created and existing specialists can be altered or deleted and this set of newly created and existing altered or unchanged specialists can be reloaded into the active workflow process instance. The orchestrator will perform the following actions in order to reload a new set of specialists:

- pause the workflow process instance
- remove all the specialists monitoring the workflow process instance zone
- load the new set of specialists that have been defined by the WF designer
- resume the workflow process instance

The functionality mentioned till this point that has been built into the orchestrator fulfils all the quality indicators stipulated by the functional aspect of workflow management.

5.6 Workflow Designer

The workflow designer is built as a client application that interfaces through a WF API to allow the creation of workflows. The steps to create a workflow are as follows:

1. Insert metadata (name of workflow and description) for workflow
2. Define constraints and dependencies for the workflow that will be checked when a new WF process is spawned
3. Define the WF process by defining the specialists for the workflow

Screenshots of how this done using WEWE's workflow designer is contained in Appendix C. A legitimate criticism can be lodged with respect to the usability of the designer -WF designers require detailed knowledge of the internal workings of the blackboard paradigm before designing any WFs. However, this research only focuses on the workflow engine and not any external workflow client applications linking to the workflow engine such as the workflow designer. The WF designer was deliberately built this way to promote experimentation and to conduct as many tests as possible which this research requires. A more user friendly and intuitive interface can be considered in the future that will abstract all implementation details of the workflow engine. This will enable the WF designer to be apathetic towards the blackboard nature of the workflow engine. An interface could be built that allows the WF designer to simply create a flowchart of the workflow like in most other WFMSs [32, 34 and 43] and the application will automatically create necessary specialists in the background.

5.7 Interaction of WF Engine Components

The previous sections of this chapter have separately explained the different components of the blackboard system within the workflow engine. This section demonstrates how each component functions and interacts with other components to provide a holistic picture. An example is used with a workflow for approving an insurance claim with five possible actions that include patterns 1 and 4 (sequence and exclusive choice; See Section 2.4 for details) as seen in Figure 32.

Recall from Section 2.2, the important differentiation between enabling and triggering a WF activity. The WF activity can execute a set of actions when it is enabled as opposed to running a different set of actions when it is triggered. In this example, the workflow process will start by sending all the claim information to the inspector when enabling the 'send to inspector and access claim' action. The 'send to inspector and access claim' action will not be executed automatically when it is enabled. It is triggered by the inspector who will either approve or reject the claim thereby executing the 'access claim' action which automatically execute the 'Notify Client (decline)' or 'Process Claim' actions by either rejecting or approving the claim respectively. If the inspector rejected the claim,

the client will be notified of the rejection and the workflow process will end. If the inspector approved the claim, the claim will be processed by executing the 'process claim' action. Depending on the software infrastructure, the claim might not be processed immediately. A periodic batch process might be running that processes all claims at the end of every day. Perhaps, the claim will be inserted into a queue that will be processed at a later time, or any number of similar processes, that will cause a delay in processing the claim. Therefore, the 'process claim' action will not automatically trigger the 'notify client (approve)' action. The 'process claim' action will only enable the 'notify client (approve)' action and this action will be executed when it is triggered by a message sent to it by an external entity that the claim has been processed. After notifying the client of approval, the workflow process will be terminated.

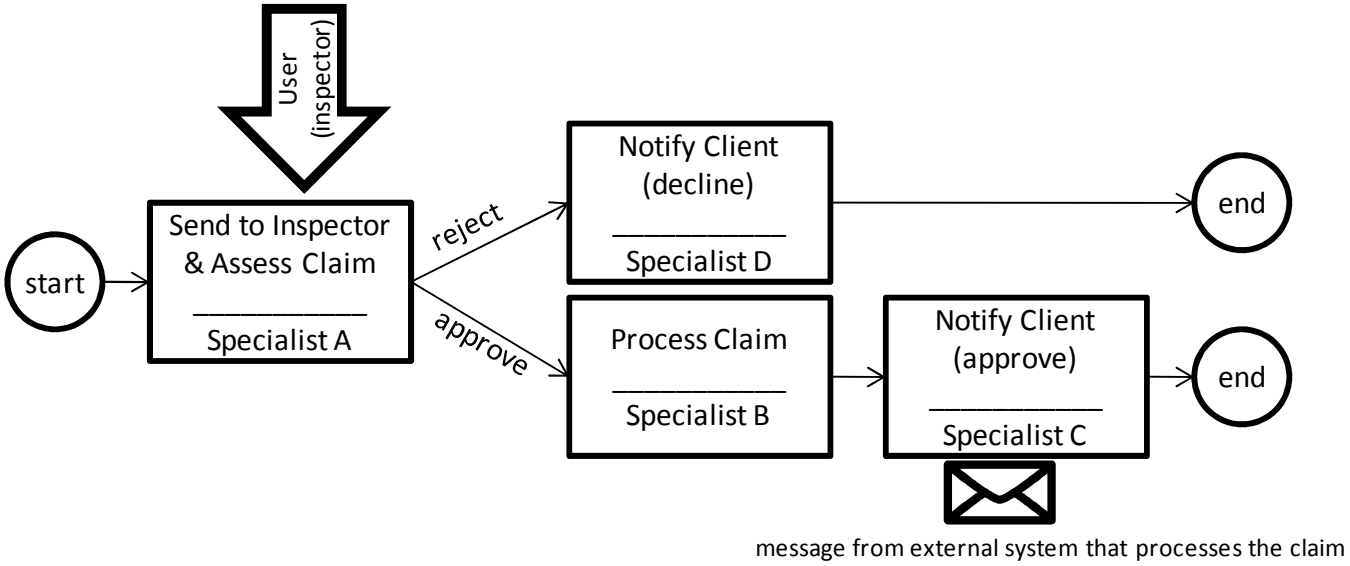

Figure 32: Hypothetical workflow of approving or rejecting an insurance claim.

Appendix D goes into great detail on how the blackboard system is used to execute this workflow. All attributes and rules for each specialist and agent are exposed for the reader to scrutinize. Snapshots of the blackboard data are shown at every stage of the workflow process and the role of each specialist and agent in the altering of the blackboard at each of these stages is discussed. For a more detailed understanding of the workings of the blackboard system, Appendix D should be consulted.

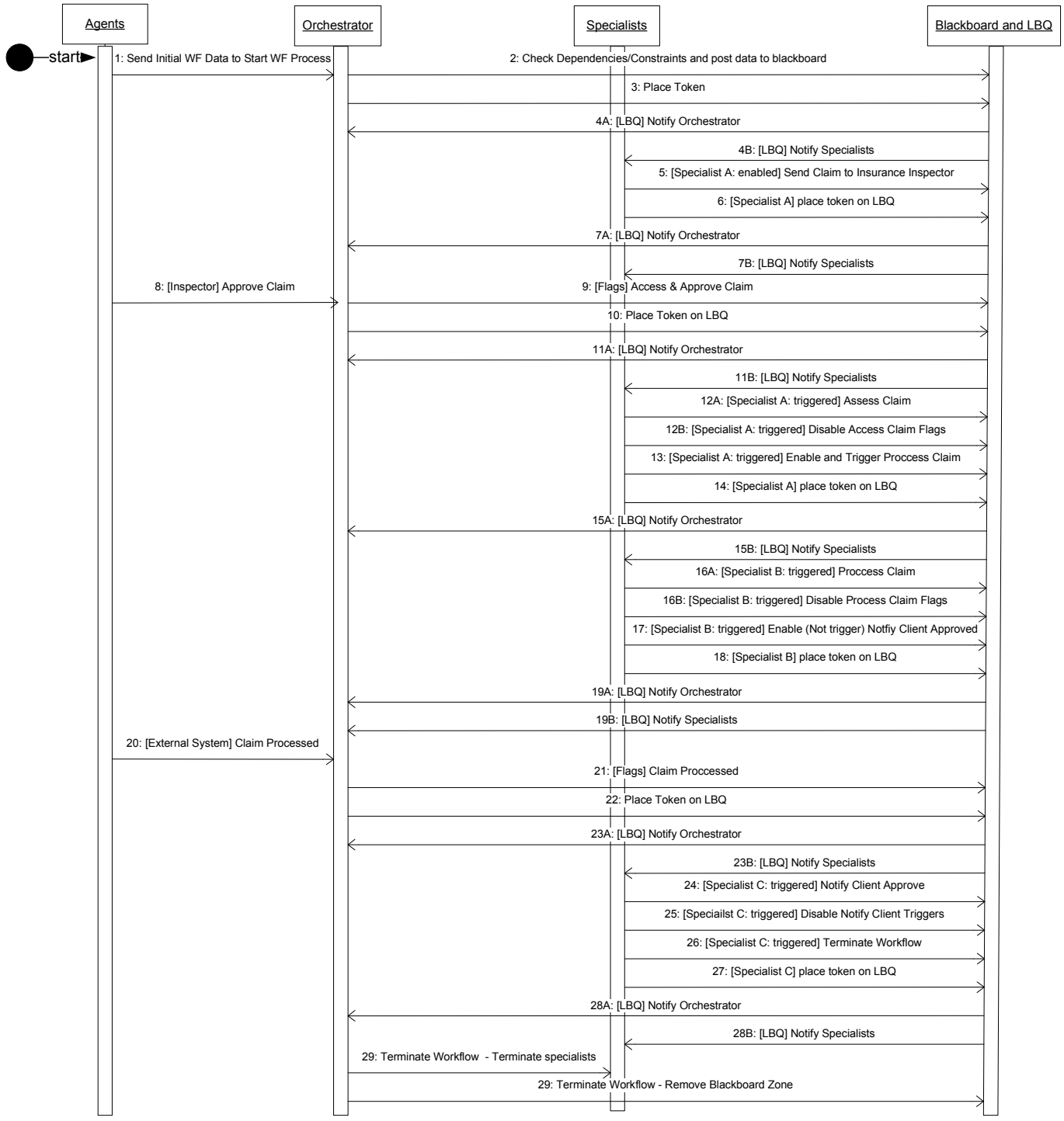

Figure 33: Sequence diagram for the workflow involved in approving and processing an insurance claim.

Figure 33 shows a high level sequence diagram of how the blackboard system that executes this hypothetical workflow. This example shows the workflow path taken when the inspector approves the insurance claim. To spawn a new workflow process instance of this workflow, the initial WF process data is brought by an agent to be posted on the blackboard. The mediator pattern as discussed in Section 5.5.5 can be seen clearly here where the orchestrator acts as an interface between incoming agents and the blackboard. Each specialist is dedicated to performing a workflow activity.

Specialists can consist of many rules (condition-action pairs). In this case, Specialist A consists of three attributes. The first attribute is for the case when Specialist A is enabled. The other two are for handling the cases when Specialist A is triggered which is when the inspector approves or declines the insurance claim.

The initial WF data is put on the blackboard and all the specialists start monitoring this data. The initial data consists of a flag which enables Specialist A. Specialist A will execute its relevant rule to send the insurance claim to the inspector. The workflow is paused and cannot proceed any further until the inspector approves or declines the claim.

The insurance inspector in this example approves the claim through a WF client application which creates an agent to be sent to the blackboard. This agent essentially consists of two flags; one to trigger the "access claim" WF action and another that indicates the claim has been approved. Specialist A monitoring the blackboard will execute its pertinent rule to access and process the claim. This rule will also enable the next WF activity to notify the client.

As soon as the actions like the accessing and processing of the claim are completed, the specialist automatically disables their respective flags. If the flag remains unchanged, the specialist will continuously execute the relevant rule since the conditions of the specialist rule are met by the existence of that flag.

The enabling and disabling of flags to execute WF actions is instrumental in realizing many WF patterns especially ones that involve branching and looping. As a simple example for looping, one can create a while or for loop by enabling a flag to execute a WF activity over and over until some condition is met and the flag is disabled to break out of the loop. Chapter 6 goes into detail on how flags and specialist rules are to be used to implement different kinds of workflow patterns.

Once the external system has completed processing the claim, it will send an agent with a flag to notify the client that the claim has been approved. This flag is placed on the blackboard and Specialist B monitoring the blackboard will execute its relevant rule. This specialist rule will perform the action of notifying the client. After that it will disable "notify the client" flag so that this activity is not repeated again. The last action in the specialist's rule is to terminate the workflow by setting the status of the workflow process on the blackboard to 'completed'. The orchestrator will see this status change and will terminate this workflow.

Whenever a specialist, agent or orchestrator makes a change on the blackboard, the entity places a token into the Linked Blocking Queue (LBQ) to notify idle specialists and the orchestrator to start monitoring the blackboard as seen in Figure 33. The LBQ in the sequence diagram is depicted to be a part of the blackboard. Despite this not being currently implemented in WEWE, it is suggested in Section 5.5.3 for the future that the LBQ be put inside the WF process object (zone) on the blackboard.

Sometimes, a situation can occur where the LBQ notifies the specialists and orchestrator but the data present on the blackboard does not allow any of these entities to engage with the blackboard. The workflow process becomes

paused until new data is posted to the blackboard via an agent. For instance, step 6 in Figure 33 shows Specialist A placing a token on the LBQ but no workflow activities are executed until new data is posted by an agent for the inspector approving a claim. Sometimes, a situation can occur where multiple specialists engage with the blackboard consecutively after each one places a token on the LBQ. For instance, step 14 in Figure 33 shows Specialist A placing a token on the LBQ after executing the 'Assess Claim' action and placing a flag to enable and trigger the 'Process Claim' action and thereafter Specialist B executes the 'Process Claim' action.

5.8 Summary

This chapter has presented how the blackboard paradigm was utilized in building a prototype workflow engine. The design and implementation of the blackboard, specialists, orchestrator and agents are provided.

The blackboard is split into zones for each WF process instance which belong to a single workflow. Information in the WF process instances can be added, changed and removed by specialists and the orchestrator/controller. Changes made to the WF process instances are saved into the database.

Specialists are designed to resemble forward-chaining rule-based systems that can encompass many rules (condition-action pairs). Specialist rules are designed and implemented to accommodate the creation of compounded antecedents with the use of the 'AND' and 'OR' conjunctions. In the context of workflow management, time-based rules (time lapsed and timeout) are also incorporated to enable time-based triggering of workflow actions. Specialist actions are designed and implemented to fulfil all requirements specified by the operational aspect of workflow management. From [16], it is stipulated that specialists should be static repositories of knowledge. Specialists follow the process illustrated in Figure 27 in order to make a contribution on the blackboard and ensuring that specialist contents remain unchanged.

The orchestrator that has been implemented has many responsibilities. It handles the spawning, management and termination of all WF processes. It keeps track of the all the WF process control threads that control every single active WF process. It is also used in the creation and termination of sub-workflows. Using the producer-consumer pattern inside the WF process control threads, concurrent specialist triggering was achieved. The functionality implemented in the orchestrator fulfils all the requirements of the functional aspect of workflow management.

Although not part of the blackboard ecosystem, agents are introduced in order to increase interoperability with external systems. Agents are able to pass data into and request data out of the workflow engine and the workflow engine can request and pass data to external systems by using agents.

The next chapter presents results from test cases developed and executed within WEWE to examine the quality indicators for the behavioural and informational aspects of workflow management through the use of workflow and data patterns respectively.

6 Analysis and Testing

6.1 Introduction

This chapter presents all test cases created and the results found for the quality indicators for the behavioural and informational aspects of workflow management. WF pattern test cases were developed and executed within WEWE to test all the 20 main workflow patterns. Support for the 40 main data patterns is also examined. The overall result of the workflow and data pattern tests for the prototype is compared with other leading workflow engine implementations.

6.2 WF Patterns

At least two test cases are created for each workflow pattern. The first test case is to create an automated workflow. Each workflow activity is enabled and triggered by the previous workflow activity, so that the workflow starts and is completed without any outside interaction. The second test case is to create an interactive workflow for each workflow pattern. In each test case, every single workflow activity is triggered by an outside entity be it a human or a machine. As discussed in Section 5.7 it does not matter what type of external entity (human or machine) is triggering workflow activities. These external entities have to create agents with flags that are posted to the blackboard. Additional test cases might include the enabling of different paths of the workflow but this is dependent on what type of workflow pattern is being tested.

For each test case, generic workflow activities are created with a logging action that logs to a text file. For instance, the log of a sequential workflow of three activities would have activity 1, 2 and 3 in this order. The logging allows execution paths to be inspected. If any one of the test cases belonging to a workflow pattern does not provide the correct execution path, then that workflow pattern is deemed to be unsupported by this workflow engine.

Using the above test cases, one can gauge whether or not individual workflow patterns are supported. However, further test cases are developed that combine individual workflow pattern test cases together and this is where test cases might fail. For instance, a parallel split (AND-split) workflow pattern test case that executes successfully by

itself might fail if the parallel split pattern is placed within the loop of the iteration pattern. Another example could be where any basic workflow pattern might fail if it is placed inside the deferred choice workflow pattern (See Section 6.2.3). The introduction of the milestone pattern (see Section 6.2.3) to any other workflow pattern might cause it to fail. For a workflow pattern to be supported, it must execute successfully if it is injected into or combined with other workflow patterns. Specific test cases are created for these kinds of scenarios.

During the creation of test cases, great care was taken to preserve the style and consistency in which specialist rules were created. Where ever possible, specialist rules were duplicated within other specialists. Moreover, entire sets of specialists were copied from one WF pattern test case to other different WF pattern test cases. This was all in an effort to simplify specialist creation within workflows. In the future, this will help creating a more user friendly GUI interface that allows users to simply create a flowchart of the workflow that is implemented in most other WFMSs and WEWE will automatically create the required specialists in the background.

The evaluation criteria used in [14, 24 and 25] are used here for evaluating WEWE. If WEWE supports a particular workflow pattern completely, a rating of "+" is given. If a workflow pattern can be partially implemented (explained in latter sections), then a rating of "+/–" is given. If a workflow pattern is not supported, then a rating of "–" is given.

6.2.1 Basic Control Flow Patterns (WF Patterns 1-9)

The basic control flow patterns as described in [14, 24] are listed in Table 1 with their support ratings. WEWE supports all basic workflow patterns.

Table 1: Workflow patterns 1-9 and their support rating.

NO	NAME	SYNONYM	DESCRIPTION	SUPPORT
1	Sequence	Sequential Actions	One activity is executed after another.	+
2	Parallel Split	AND-Split	A single thread splits into two threads that are executed in parallel.	+
3	Synchronization	AND-Join	Two threads of control join into one thread of control. Moreover, both incoming threads have to be completed before joining into a single thread.	+
4	Exclusive Choice	XOR-Split	Depending on the decision, only one thread of control can be chosen to be executed from several prospective threads of control.	+

NO	NAME	SYNONYM	DESCRIPTION	SUPPORT
5	Simple Merge	XOR-JOIN	Multiple threads of control join into one thread of control without synchronization. The joining thread is executed whenever an incoming active thread is completed. It is assumed that multiple threads of control are never executed in parallel.	+
6	Multi-Choice	OR-Split	Depending on the decision, one or many threads of control are chosen to be executed in parallel from a single thread of control.	+
7	Synchronizing Merge	Synchronizing join	Multiple threads of control join into one single thread of control with synchronization. The joining thread is executed only when an/all incoming active thread/s is/are completed.	+
8	Multi-Merge	Non-Synchronizing Join	Multiple threads of control join into a single thread of control without synchronization. The joining thread is executed every time an incoming thread is completed.	+
9	Discriminator	N of M Synchronizing Join	Only N out of M multiple threads of control join into a single thread with synchronization. Once N incoming threads are completed, the joining thread is executed and all other incoming threads that are not completed are discarded.	+

Workflow patterns 2, 4 and 6 are similar in nature in that they split a single WF thread into one (XOR-split), two (AND-split) or many (OR-Split) threads that are executed concurrently. The only difference in these WF patterns is the specialist rules that enable one, two or more branches. A special focus was placed on the execution of concurrent WF threads and how the workflow engine handles synchronization of shared data on the blackboard. Tests were conducted for all three WF patterns, but WF pattern 2 is chosen for the purposes of this discussion. Figure 34 shows a WF pattern of an AND-split where after WF activity A, the WF splits into two threads that are executed in parallel.

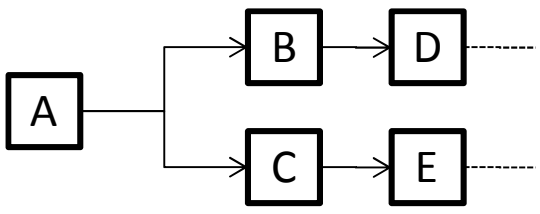

Figure 34: The AND Split WF pattern.

A specialist exists for every single WF activity (A, B, C, D and E). Specialist A will execute WF activity A and then enable the two branches. Thereafter, both specialists B and C will execute their attributes concurrently executing activities B and C concurrently. Then specialists D and E will execute their rules and so on. There are no issues when multiple threads (specialists) read shared data but a race hazard might occur when multiple threads (specialists) operate on, or try to alter, the same data concurrently. To investigate this issue, a variable called 'action_trace' was placed on the blackboard and each specialist was created to have an action that concatenates their moniker (alias) to this variable. So at the end of the workflow, the 'action_trace' variable could only have four possible values depending on how the specialists interacted with blackboard which are ABCDE, ACBDE, ABCED and ACBED. Both specialists B and C or D and E are trying to concatenate their moniker at the same time to the 'action_trace' variable but access is synchronized to variables in cases when variables are being updated and deleted. To detect whether or not there is a race hazard specifically between specialists B and C or D and E, this workflow pattern was executed 500 times and it was found that 44% of times specialist B was executed before specialist C which is close to 50%. This suggests two specialists that want to change the same shared data have an equal opportunity to do so. Furthermore, the execution order of two concurrent actions does not bias the execution order of subsequent concurrent actions. It was found that 47% of time execution times action E was executed before D (almost 50%).

It can be noticed that WF pattern 7 is just an extension of WF pattern 2. The AND join merges only two incoming threads while the synchronizing merge can merge many threads. The implementation of WF pattern 2 and WF pattern 7 was the same in WEWE with the only difference being in the specialist rules that handle the merging of incoming branches. The specialist that handles the synchronizing merge consists of a single compound rule that checks whether or not the last WF activity of each incoming WF thread has been executed. In the context of the blackboard paradigm, a specialist rule has to exist to monitor the flags to indicate whether or not the last WF activity of each incoming branch has been executed. The number of incoming branches dictates how many antecedents have to be included into the specialist rule. The antecedents are conjoined together with AND rule conjunction.

Similarly, WF pattern 8 is an extension of WF pattern 5. The former assumes that only one thread can be active while the latter can handle multiple active threads. WEWE's implementation of these two patterns is similar as well. The specialist that handles the non-synchronizing join of multiple incoming branches is similar to a specialist that handles the synchronizing join on incoming branches in WF patterns 2 and 7. The only difference is the specialist antecedents are joined together with ORs (rule disjunctions) instead of ANDs (rule conjunction).

Workflow patterns 2, 4 and 6 split workflows into multiple threads of control. Each one of the basic control flow patterns were injected into new threads of control produced by these workflow patterns in order to test their behaviour. For instance, the parallel split pattern was injected inside one of the multiple threads of the parallel split pattern. Another test case was created where one of the threads produced by the exclusive choice pattern is split

further by the multi-choice pattern and merged later by the multi-merge pattern. A total of 32 test cases were created where individual workflow pattern test cases were combined together to create new test cases for more rigorous testing.

The majority of mainstream workflow engines described in [14, 24 and 25] support all the simple control flow patterns as does WEWE.

6.2.2 Structural Patterns (WF Patterns 10-15)

WF patterns 10 to 15 are grouped as structural patterns since other patterns such as simple control flow patterns (WF patterns 1 to 9) can be executed within them. Majority of the structural patterns are different kinds of loops. All the simple control flow patterns (WF patterns 1-9) can be executed within these different loops or structural patterns. Therefore, in order to test whether or not these structural patterns are supported, every single test case made for WF patterns 1 to 9 was run inside the looping patterns giving a total of 75 test cases. If any of the simple control flow patterns (WF patterns 1-9) exhibited unexpected behaviour within the structural patterns, then support for that structural pattern can be deemed as partial or no support at all, depending on how many simple control flow patterns fail within that structural pattern. Table 2 lists all the structural patterns and their support ratings for WEWE.

WF pattern 10 can be seen to be as a GOTO statement. The majority of mainstream workflow engines as found in [14, 24 and 25] do not provide direct support for this WF pattern. These workflow engines typically implement this WF pattern by converting a GOTO statement into a structured loop. However, WEWE fully supports this WF pattern. In the blackboard paradigm, every specialist is dedicated to executing a single WF activity. The specialist will only execute its WF activity (the WF activity can be composed of many WF actions) if a trigger flag exists on the blackboard. In addition to the WF activity, the specialist would also have actions to enable or trigger the next WF activity by placing an appropriate flag to continue the WF process. GOTO statements in the context of the workflows have the ability to jump to any state of the workflow regardless of whether or not that state is after or before the current state within the workflow path. For a GOTO statement to be a loop, the GOTO statement must always jump to a previous WF state in order to repeat WF activities. In the blackboard paradigm, at any point in the WF, specialists can consist of extra rules (condition-action pairs) that re-enable flags of previous WF activities thereby creating arbitrary loops. WEWE fully supports WF pattern 10.

WF pattern 12 can be described as a situation where the same activity or set of activities is executed in separate threads independent of each other without synchronization. Instead of repeating the set of activities linearly (one after another) within a normal conventional loop, separate threads are spawned so multiple instances of the same set of activities are executed concurrently. In this pattern, no synchronization is required. The WF can continue without having to wait for any of these independent threads to complete. Each independent thread requires some private control data (flags to trigger or enable activities, etc) that only it can access. In this prototype, this is not possible as

all data can be globally accessed by any thread or specialist. Therefore, this workflow pattern cannot be implemented. One can partially implement this WF pattern by using an OR-Split (WF pattern 6) to produce separate threads that each separately contain the same set of activities to execute. This solution can be used if a few instances of the same set of activities need to be executed but it becomes too cumbersome if many instances need to be spawned.

Table 2: Workflow patterns 10-15 and their support rating.

NO	NAME	SYNONYM	DESCRIPTION	SUPPORT
10	Arbitrary Cycles	Loop or go to statement	One or more activities are executed repeatedly	+
11	Implicit Termination		A workflow process should be terminated when there are no more activities to be executed. If a final node or final activity is reached, then the workflow process should terminate regardless of any uncompleted activities that might be running.	–
12	Multiple instances without synchronization	Spawn off	Multiple instances of a set of activities are spawned running in their own independent threads without any synchronization.	–
13	Multiple instances with a priori design time knowledge	For loop	A set of activities are executed multiple times. The number of times the set of activities are executed within a loop is stipulated at design time.	+/–
14	Multiple instances with a priori runtime knowledge	For loop	A set of activities is executed multiple times. The number of times the set of activities is executed within a loop is determined during runtime prior to start of the loop based on certain parameters defined during design time.	+/–
15	Multiple instances without a priori runtime knowledge	While loop	A set of activities is executed multiple times. The number of times the set of activities is executed within a loop is not determined during design time and not determined at runtime prior to start of the loop. While the loop is being executed, an exit condition is determined.	+

The lack of support for WF pattern 12 in the prototype does not necessarily mean that a blackboard system cannot support this WF pattern. The prototype already consists of a blackboard that is split into many zones to store data for every single WF process instance. The blackboard can be further split in local thread data zones to store data for each independent thread whenever they are created on the fly; a temporary local thread data zone is created when

an instance of a thread is created and destroyed when the thread is completed. This functionality can be built into the blackboard system in order to support WF pattern 12 whilst still conforming to the blackboard paradigm. For now, a rating of "–" is given to this prototype for its lack of support of WF pattern 12.

WF patterns 13 and 14 involve the looping of WF activities within a WF. The difference between the two WF patterns is when the limit as to how many times the loop is repeated is set. In WF pattern 13, this limit is set during design time, while in WF pattern 14, it is set during runtime. Any number of combinations of basic control flow patterns (WF patterns 1-9) can be executed within loops. To create a loop encompassing any combination of basic control flow patterns, two specialists need to be created that handle the entry into and exit out of a loop. Figure 35 demonstrates this by showing an example of a combination of the AND-split followed by an AND-join WF pattern that is within a loop.

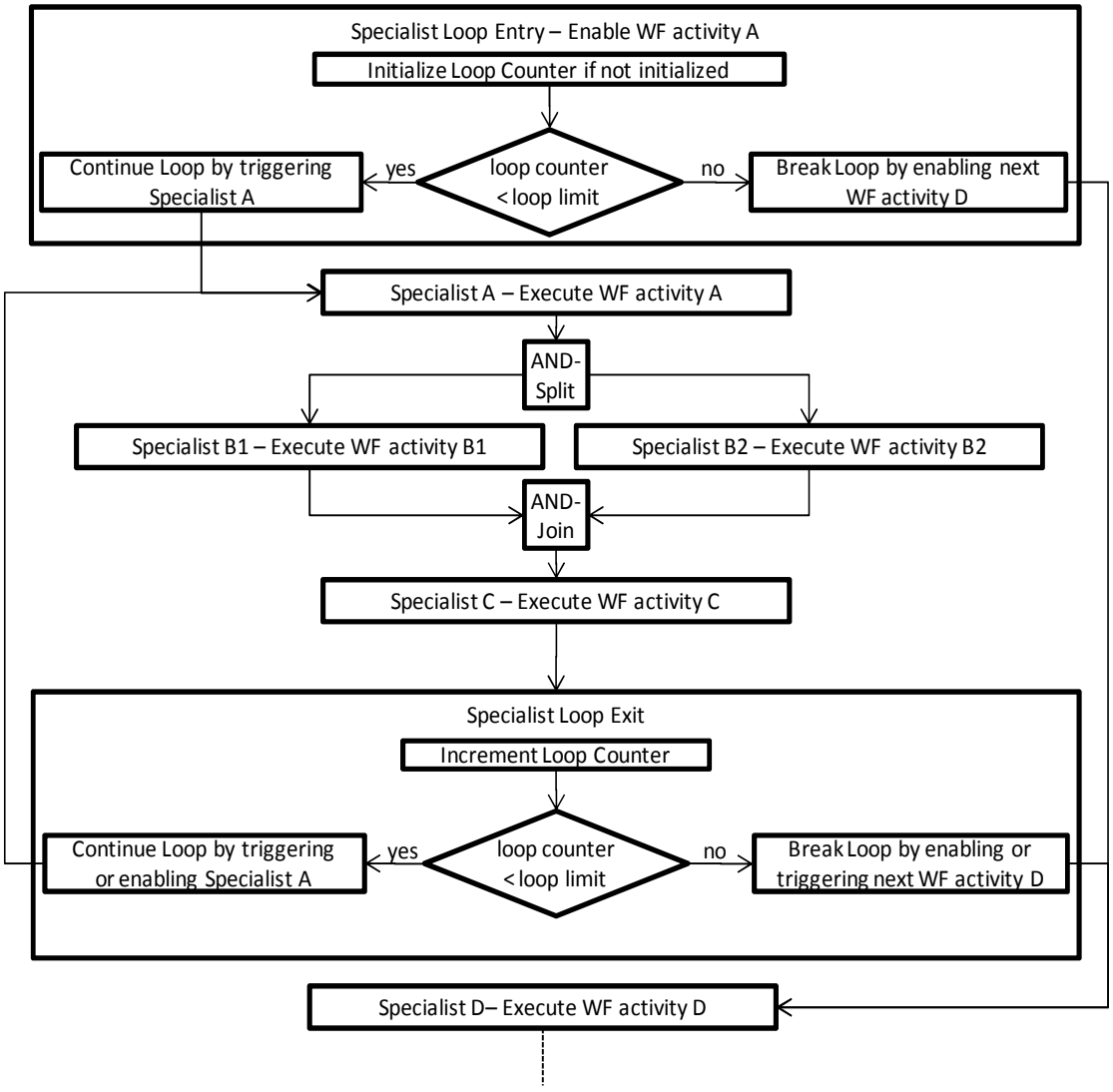

Figure 35: Example of looping using the blackboard paradigm.

Two specialists exist, one to handle entry into the loop and the one to handle the exit out of the loop. The specialist that handles the entry into the loop simply initializes the loop by initializing the loop counter if it is not initialized. This initialization happens only once the first time the WF enters the loop. So, if this loop needs to be rerun again at any time, another specialist or agent simply needs to put a flag on the blackboard to trigger the loop entry specialist thereby reinitializing and restarting the loop.

The specialist that handles the exit out of the loop increments the loop counter and performs a Boolean expression check to either repeat the loop again or to exit out of the loop. Depending on the Boolean expression, the loop exit specialist will either trigger the specialist in charge of executing the first WF activity within loop (Specialist A) thereby repeating the loop again or it will trigger the next activity in the WF (Specialist D) thereby breaking out of the loop. If one ignores the contents of the loop entry specialist, a do-while loop emerges. The loop entry specialist allows entry into the loop and it is only the loop exit specialist that either repeats or breaks out of the loop. This is not desirable as this type of loop construct (do-while) will always execute the loop at least once regardless of the initial conditions. It must also contain conditions that exist within the loop exit specialist that are able to break out of the loop. This will allow for situations where the loop can be skipped completely if the loop limit is set to zero.

As mentioned earlier, the chief difference between WF patterns 13 and 14 is when the loop limit is set which is during design and runtime respectively. For WF pattern 13, the loop limit can be set as shown in the upper part of code listing 9 to run a hypothetical loop exactly five times if the 'loop limit' variable is initialized to be five before the entry into the loop. The limit is set to be five during design time and cannot be changed.

```
//For WF pattern 13
       If (var.loop_limit <= 5)

//For WF pattern 14
       If (var.loop_limit <= var.loop_limit_actual)
```

Listing 9: Sample code for specialist rules for WF patterns 13 and 14.

An example of how the dynamic loop limit can be set is seen in the lower part of code block 9 for WF pattern 14. The loop limit is not hardcoded and is represented by a variable that can be set at runtime. It is even possible for specialists and agents to alter this variable while the loop is running.

Twenty-four test cases were created to test WF patterns 1-9. The basic control flow patterns were injected into workflow patterns 13 and 14 to test looping in each one by adding the two loop entry and exit specialists ultimately creating an additional 48 test cases to test just these two workflow patterns. All 48 test cases passed and exhibited the expected results. However, the ability to loop WF activities linearly (one after another) was only tested. WF patterns 13 and 14 also include the ability for 'spawn off' (WF pattern 12) where instead of looping activities

linearly, multiple instances of the same activities can also be spawned to be completed concurrently which was found to be not supported. The lack of support for WF pattern 12 has a negative knock-on effect into WF patterns 13 and 14 earning a rating of "+/–" for the workflow engine.

A significant advantage of a blackboard based workflow engine is evident in the implementation of WF pattern 15. Most workflow engines critiqued by [14, 24 and 25] cannot implement this WF pattern. As already stated in Section 2.8.6, these workflow engines are WF activity based and abstract from WF states. This workflow pattern specifically requires WF states. WF pattern 15 can be seen to be a while loop. A while loop consists of a Boolean expression that evaluates the state of the loop and then either repeats or breaks out of the loop. The previous workflow patterns which can be modelled as 'for' loops simply evaluate a Boolean expression that always contains a comparison between the loop counter and a loop limit. In a while loop, this Boolean expression can be more general. A workflow can be created that loops through a set of activities indefinitely until a user or external system sends an agent to place a flag on the blackboard and a Boolean expression can be formed that simply checks the existence of this flag in order to break out of the loop. This Boolean expression can be more complex and can contain a combination of antecedents to check for any number of things depending on the requirements of the workflow. To create a WF pattern of this nature, two loop entry and exit specialists have to be created similar to what is seen in Figure 35. But, the Boolean expressions in both these specialist can be more general. Additionally, the loop exit specialist contains an action to increment the loop which can be changed to any other general activity that updates the state of the loop or can be completely removed depending on the WF requirements. This WF pattern was tested and is fully supported by the workflow engine.

The implicit termination pattern (WF pattern 11) cannot be supported by the blackboard paradigm. This WF pattern describes that a workflow process should be terminated when there are no more activities to be executed. It is interesting to note that most of the workflow engines critiqued by [14, 24 and 25] support this workflow pattern while WEWE does not and this highlights the main conceptual difference between WEWE and most other workflow engines. The workflow engines that can support this WF pattern are activity based. A task manager with the workflow engine transverses through and executes WF activities until the final node is reached and at which time the task manager terminates the workflow. In WEWE, WFs are not activity based and WF states drive workflows. The final node or the last activity to execute can never be known in WEWE. Therefore, a rating of "–" is given for this WF pattern.

6.2.3 State Based Patterns (WF Patterns 16-18)

While most traditional activity based WF engines struggle to support state based WF patterns, WEWE due to its explicit incorporation of WF states within the blackboard paradigm can easily support this kind of WF patterns. WEWE provides full support. Table 3 lists all the state based patterns and their support ratings of WEWE.

Table 3: Workflow patterns 16-18 and their support rating.

NO	NAME	SYNONYM	DESCRIPTION	SUPPORT
16	Deferred Choice	Deferred XOR-split	Similar to the XOR split except that a branch is not chosen immediately but is chosen based on data or stimuli provided by the environment.	+
17	Interleaved parallel routing	Unordered Sequence	Activities are executed in arbitrary order and the order is decided at runtime based on stimuli provided by the environment. Furthermore, two activities cannot be executed at the same time.	+
18	Milestone	State condition	Activities cannot be enabled or executed if a certain state exists within the workflow process.	+

WF pattern 16 is an extension of WF pattern 4 and can be explained by the use of an example shown in Figure 36. This workflow pattern enables all prospective branches by enabling all WF activities B, C and D and there is a 'race' between these activities. All of these activities are waiting to be triggered by some external or internal stimuli. WF activities B, C and D are triggered by a human user through a WF client system, an external system by sending a message and a timeout condition respectively. For activities B and C, an agent posts data to the blackboard to trigger these two WF activities while a timeout is registered within the orchestrator which in turn triggers the specialist responsible for executing WF activity D when the timeout occurs. Depending on which external or internal stimulus triggers its WF activity first, the specialist responsible for that WF activity will run its rules. As part of its actions, it will disable the other enabled activities hence disabling the other prospective branches and enable the next WF activity hence continuing the workflow. The other external or internal stimuli that arrive later into the blackboard system will have no impact on the workflow since the WF activities have already been disabled by the WF activity that was already successfully executed.

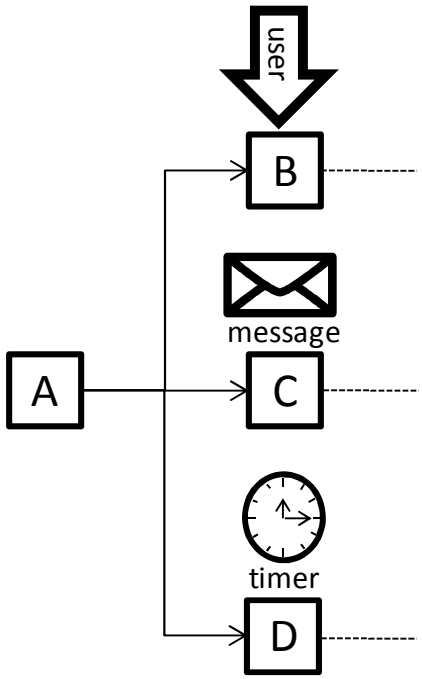

Figure 36: Example of WF pattern 16.

WF pattern 17 requires the explicit notion of WF states to be supported fully. Consider Figure 37 (a) that shows an example of WF activities B and C to be executed in any order. Those traditional workflow engines that abstract from WF states will try to use more structured workflow patterns such as differed choice and simple merge to execute this kind of WF pattern as seen in Figure 37 (b). This example has only two WF activities. For three WF activities, a further two differed choice splits need to be added within each differed choice branch. Ultimately, the WFs being created using this method will result in sprawling spaghetti like WF models if many workflow activities are required to be executed inside an interleaved sequence which is undesirable.

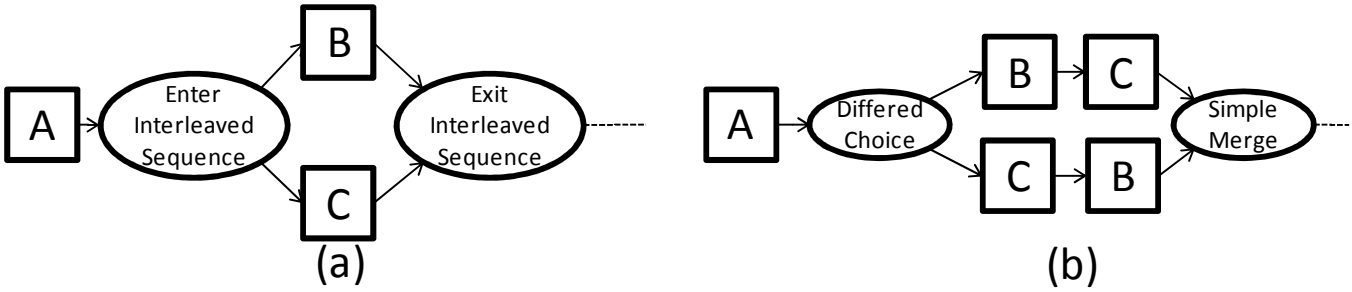

Figure 37: Example of WF pattern 17. **(a)** Direct implementation of WF pattern 17. **(b)** Indirect implementation of WF pattern 17.

In WEWE, WFs with this WF pattern can easily be created. Similar to structured loops, two specialists have to be created to handle the entering and exit out of this WF pattern (interleaved sequence). Both WF activities are enabled when the WF enters the interleaved sequence (WF pattern 17) by the interleaved sequence entry specialist. If a trigger flag gets placed on the blackboard to execute any of the WF activities inside the interleaved sequence,

the specialist responsible will execute them. But before executing the active interleaved sequence WF activity, the specialist will disable all the other interleaved sequence activities (changing flags on the blackboard) and will subsequently re-enable them after the WF activity is executed. This will prevent multiple WF activities inside an interleaved sequence to be executed at the same time as the WF pattern requires. The interleaved sequence exit specialist is constantly checking the blackboard to determine if all the WF activities inside the interleaved sequence have been executed just by checking the existence of all the trigger flags of all the WF activities. If all the triggers flags are found on the blackboard, the interleaved sequence exit specialist will disable all the WF activities within interleaved sequence so that it can possibly be run again in the future (resetting the WF pattern) and enable the next WF activity or set of activities hence exiting out of this WF pattern. WEWE fully supports this workflow pattern.

WF pattern 18 is possibly the easiest WF pattern to implement with WEWE which is notable considering it is the most difficult WF pattern to implement for most traditional activity based workflow engines. This WF pattern can be simply described by a scenario where enabling or triggering of a workflow activity or activities depends on the WF process being in a specific WF state or succinctly put, when the WF reaches a certain milestone. The WF state or milestone can be reached if a certain specified number of WF activities have already been executed. Figure 38 (a) shows an example of workflow where a WF activity cannot be enabled unless a WF milestone has been reached. The WF milestone is reached when WF activity A has been executed. Some other workflow engines try to use differed choice with the multi-merge WF pattern to implement the milestone WF pattern. This example however has only one WF activity A to be executed as a milestone. With multiple WF activities, an OR-split followed by many branches of Merge and Differed Choice patterns conjoined together (one of each workflow activity as shown in Figure 38 (b)) and a synchronizing join needs to be created. Again, a sprawling spaghetti like WF model has to be created for traditional workflow engines in order to try to implement this workflow pattern which is undesirable.

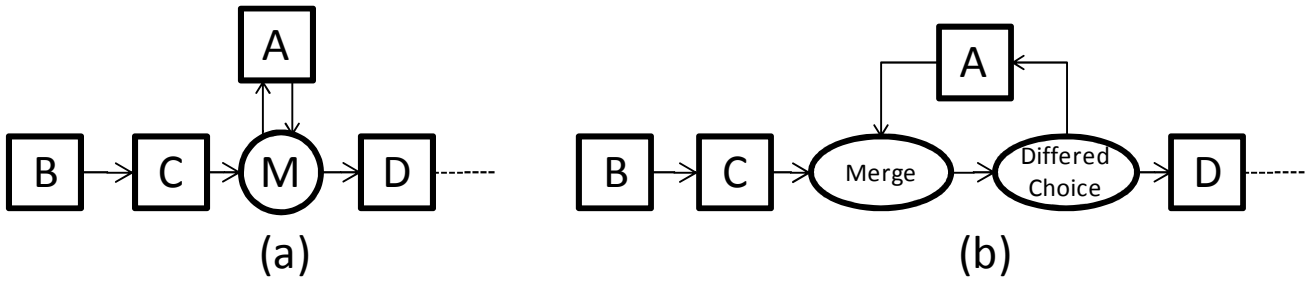

Figure 38: Example of WF pattern 18. **(a)** Direct implementation of WF pattern 18. **(b)** Indirect implementation of WF pattern 18.

In WEWE, one specialist has to be created with a single attribute (rule-action pair) that checks that certain specified WF activities have been triggered for a milestone to be reached and puts a milestone flag on the blackboard. Other specialists responsible for executing WF activities considering this milestone can simply have an added condition within their rules to check the existence of this milestone flag. WEWE fully supports this workflow pattern.

6.2.4 Cancellation Patterns (WF Patterns 19-20)

The cancellation patterns are fully supported and can easily be implemented by WEWE. Table 4 lists all the cancellation patterns and their support ratings of WEWE.

Table 4: Workflow patterns 19-20 and their support rating.

NO	NAME	SYNONYM	DESCRIPTION	SUPPORT
19	Cancel Activity	Withdraw Activity	An enabled activity is disabled and skipped.	+
20	Cancel Case	Withdraw case	An entire workflow process instance or case is removed. Even if some WF activities are currently running, they are terminated and the WF process is terminated.	+

WF pattern 19 is very similar to WF pattern 16 (Differed Choice) where one workflow activity cancels another by disabling an enabled WF activity. For WF pattern 19, an external entity outside of the workflow engine can cancel the activity (disabling an enabled WF activity) by using an agent. A WF activity can be cancellable if an additional specialist rule (condition-action pair) that handles the cancelling process is added to the specialist that is responsible for executing that cancellable WF activity. When a WF activity cancel flag is posted to the blackboard by an agent, the specialist rule that handles the cancellation process will be executed that simply disables the enabled WF activity and enables or triggers the next WF activity thereby continuing the WF process.

The ability to cancel an entire WF process can be set at any point in the WF inside WEWE. A specialist rule (condition-action pair) can be added to any specialist within the workflow that can cancel the WF process instance at any point. The condition of this cancel WF activity rule could be based on the existence of a milestone or any other kind of flag or combinations of flags present on the blackboard. To cancel the workflow process, the specialist simply changes the workflow status on the blackboard from ‘active’ to ‘cancelled’. The change in workflow status will prompt the orchestrator to terminate the WF process.

6.2.5 Final Results and Behavioural Comparison with Existing Workflow Engines

There are 20 workflow patterns. If one takes a rating of “+” (full support) to be a score of 1, rating of “+/–” (partial support) to be a score of 0.5 and a rating of “–” (no support) to be a score of 0, a score of 17/20 (85%) can be given to WEWE.

The final score of WEWE can be increased from 17/20 (85%) to 19/20 (95%) if the ability to ‘spawn off’ WF activities is added to this prototype. As mentioned earlier in Section 6.2.2, this can be done easily whilst not breaking the blackboard paradigm. Ultimately, the only WF pattern that WEWE cannot support is WF pattern 11 (Implicit Termination). The nature of the blackboard system makes it impossible to support this WF pattern.

Besides this, all other WF patterns can be implemented, including some with great ease, to provide direct and complete support of the WF patterns.

These workflow patterns determine the level of functionality and provide a way to compare WFMSs. Van der Aalst in [14] did a comparison of 15 leading workflow engines in 2003 and found that 51.6% of patterns could be implemented. The same study was performed by [24] (Van der Aalst and ter Hofstede) in 2006 with new versions of some of the same workflow engines and some new ones (14 workflow engines in total) and found that 58.4% could be implemented. The same authors of [14] and [24] have provided a website called the "workflow patterns initiative" in which they have continued their study [125]. They have applied their workflows pattern tests to new versions of the same workflow engines and some new ones (17 workflow engines in total) as of 2013 and found that 64.1% of the workflow patterns could be implemented. A gradual improvement in support can be seen.

It is worthy to mention that no workflow engine in this collection can support all the 20 workflow patterns. Furthermore, it is stated in [111] that many of these workflow management systems were designed and developed to focus on maximum support of these workflow patterns. jBPM and BPMN were designed and developed to focus on support of these workflow patterns and they both can implement 87.5% of the workflow patterns [17]. YAWL (see Section 2.8.4 for implementation details) is another workflow engine that is specifically designed for maximum support of workflow patterns and it can support 95% of all the main workflow patterns [47]. The only workflow pattern that is not supported by YAWL is the implicit termination pattern. YAWL does not use the blackboard approach. However, like WEWE, it is interesting to note that the creators of YAWL also use Petri-nets as the starting point to building their workflow engine and both WEWE and YAWL have identical support for WF patterns.

In contrast, the focus of this research was to develop a workflow engine using the blackboard paradigm stringently following the guidelines and stipulations provided by [16] in order to determine how well such an approach can support these workflow patterns. Empirically, it has been determined that implementing a WF engine using a pure blackboard paradigm in not only viable, but successful in that the behavioural functionality that is achievable matches or exceeds most other approaches.

6.2.6 Implementation Comparison with Existing Workflow Engines

Many workflow engines have been investigated and mentioned in Section 2.8. Each workflow engine investigated has its own distinctive architectural style and implementation of various modelling techniques. Moreover, all of them did not use any one single architectural pattern but consisted of heterogeneous architecture. This is in strong contrast to the workflow engine for this research which is built using solely the blackboard paradigm.

A common attribute of most workflow engines is that they abstract from the state of the workflow and focus on the workflow process rules and actions (see Figure 17). They focus more on keeping track of the transitions between workflow activities in order to know what activities to execute next within a workflow process. They transition

from one set of activities to be executed to the next until the workflow process is completed. Different workflow engines accomplish this in various ways as already mentioned in Section 2.8. According to [14] and [24], workflow management systems that do not explicitly incorporate workflow process state and data cannot implement some of the complex workflow patterns.

By contrast, a workflow engine built on the blackboard paradigm is not simply activity based (abstracted from workflow states). It includes not just the workflow activities (actions) and workflow rules (transitions) but also the workflow data to determine the workflow state (see Figure 17 for the relationship between these entities). Furthermore, all of these three entities that make up the workflow process (traditional workflow management systems only include two of the three entities by discarding workflow state or data) fit neatly into the blackboard paradigm as seen in Figure 18. By factoring in workflow states into a workflow management system built using the blackboard paradigm, complex workflow patterns can be implemented.

6.3 Data Patterns

While WF patterns test various control flow which defines the ordering of activities that need to be executed and event handling, data patterns focus on data flow which aims to provide correct data at the right time so any activity in a workflow can be executed successfully. By creating test cases for WF patterns, one has already indirectly tested most data patterns. The WF pattern test cases revealed that all the WF activities can be successfully executed because the availability of WF data at the right time that is required to execute any WF activity. Nevertheless, data patterns are explicitly tested in order the fulfil requirements of the informational aspect of workflow management as stipulated in Section 4.3.2.

Similar to WF patterns, it needs to be determined how many of the 40 different data patterns are supported by WEWE. The same rating system as used in WF patterns will also be used for data patterns. Many WF pattern test cases already capture the essence of many data patterns. These test cases were reused and support for some of these data patterns was verified by inspection when executing these test cases. For specific complex data patterns, additional WF test cases were developed that capture the essence of these data patterns. The flow of data was observed when these test cases were executed to determine whether or not a particular data pattern is supported.

6.3.1 Data Visibility Patterns (1-8)

These sets of data patterns focus on how elements are generated and utilized by different components of the workflow engine. As can be seen in Table 5, some patterns are fully supported while others have either partial or no support.

Table 5: Data patterns 1-8 and their support rating.

NO	NAME	DESCRIPTION	SUPPORT
1	Task Data	WF data that is generated by WF activities are only accessible to those WF tasks.	+
2	Block Data	WF data that is generated and required by block tasks or WF activities inside a sub-workflow are accessible to any component of sub-workflow.	+
3	Scope Data	WF data can be created which is accessible by a subset of WF activities within a WF process.	+
4	Multiple Instance Data	WF data can be used and generated for every execution instance by a set of tasks that are executed multiple times.	+/–
5	Case Data	WF data can be accessed by any component of the workflow at any time in the execution of the WF process instance.	+
6	Folder Data	Common data can be created and accessed by multiple WF process instances of the same kind of WF.	–
7	Workflow Data	Common data can be created and accessed by all components of every WF and WF process instance within the workflow engine.	–
8	Environment Data	Data elements from the external operating environment can be accessed by any component of the workflow.	+

WEWE invokes application program plug-ins to execute workflow tasks that are essentially Java programs. Data that is generated and used in these Java programs can be restricted by not placing this data on the blackboard. This enables support for data pattern 1.

Considering data accessibility of different components of a workflow process such as block tasks (data pattern 2), sub-workflows (data pattern 2) and workflow activities (data pattern 3) is unnecessary in the context of the blackboard paradigm. Data on the blackboard is always accessible to any component of a workflow process at anytime. Therefore, data patterns 2 and 3 are fully supported.

Data pattern 4 is partially supported because the prototype system does not have the ability for WF activities to 'spawn off'. If WF pattern 12 (spawn off) is implemented, then this data pattern can be fully supported also.

Data pattern 5 is fully supported in the blackboard paradigm. Specialists created for a workflow process instance or case only monitor the blackboard zone that houses data for that particular case. This allows any component of the workflow process to access data from that blackboard zone at anytime.

In this prototype, blackboard zones for housing folder and workflow data is not created. Therefore, data patterns 6 and 7 are not supported. There is no definitive solution to fully support the access to all environment data (data pattern 8). A specialist executes WF actions by executing application programs. These JAVA based application programs can access environment data. A simple application program was developed to append to text file which demonstrates that environment data (text file within file system) can be accessed. Also, a specialist can invoke an agent to request data from the operating environment external to the workflow engine.

6.3.2 Internal Data Interaction Patterns (9-14)

Interaction data patterns focus on how WF data can be created and passed between components (WF activities) in a workflow process. As can be seen in Table 6, some patterns are fully supported while some others have partial support.

Table 6: Data patterns 9-14 and their support rating.

NO	NAME	DESCRIPTION	SUPPORT
9	Data interaction between WF activities	Required WF data can be passed from one WF activity to another.	+
10	Data interaction – block task to sub-workflow decomposition	WF data can be passed from a block task (group of WF activities) to a sub-workflow when it is spawned and is being executed.	+
11	Data Interaction – Sub-workflow Decomposition to Block Task	WF data can be passed from a sub-workflow to the block task (group of WF activities) once the sub-workflow is completed.	+
12	Data Interaction – to Multiple Instance Task	WF data can be passed from a preceding WF activity to the next WF activity. Additionally, WF data can be passed to all the multiple instances of the same multiple instance task.	+/–
13	Data Interaction – from Multiple Instance Task	WF data that is generated by multiple instance tasks can be passed to the next WF activity.	+
14	Data Interaction – Case to Case	WF data can be passed from an active WF process instance to another concurrently running WF process instance.	+

Data patterns 9, 10, 11 and 12 are fully supported by WEWE. Discussions around how WF data is passed between different components are basically moot. All data is placed on the blackboard and no explicit data passing is required. WF components (specialists and agents) can access any data on blackboard when they require it.

Again the lack of support of WF pattern 12 has a negative permeating affect on some data patterns that are involved in some sort of looping and particularly the spawning of multiple concurrent WF activities. Data pattern 12 is partially supported since the WF engine does not have capability of WF activity 'spawn off'.

WF pattern 14 is fully supported through the use of agents that can pass data from one blackboard zone to the next. A specialist can invoke an agent to pass any WF data from its WF process instance to any other WF process instance on the blackboard.

6.3.3 External Data Interaction Patterns (15-26)

External data interaction patterns deal with data being exchanged between components of the workflow process instance and some external resource or service outside the realm of the workflow engine. As seen from Table 7, the first eight data patterns are fully supported while the last four are not supported.

Table 7: Data patterns 15-26 and their support rating.

NO	NAME	DESCRIPTION	SUPPORT
15	Data Interaction – Task to Environment – Push-Oriented	WF data can be pushed from the WF engine to a resource or service in the operating environment.	+
16	Data Interaction – Environment to Task – Pull-Oriented	WF data can be requested from a resource or service in the operating environment to execute a workflow task.	+
17	Data Interaction – Environment to Task – Push-Oriented	WF data can be received from services and resources in the operating environment and utilized during the triggering or execution of a WF task.	+
18	Data Interaction – Task to Environment – Pull-Oriented	A WF task has the ability to receive and respond to requests for WF data from the services and resources in the operating environment.	+
19	Data Interaction – Case to Environment – Push-Oriented	A WF process instance has the ability to pass WF data to services and resources in the operating environment.	+

NO	NAME	DESCRIPTION	SUPPORT
20	Data Interaction – Environment to Case – Pull-Oriented	A WF process instance has the ability to request data from a service or a resource in the operating environment.	+
21	Data Interaction – Environment to Case – Push-Oriented	A WF process instance has the ability to accept any WF data that is passed from a service or a resource in the operating environment.	+
22	Data Interaction – Case to Environment – Pull-Oriented	A workflow process instance has the ability to respond to requests for WF data from a service or a resource in the operating environment.	+
23	Data Interaction – Workflow to Environment – Push-Oriented	The ability for a workflow engine to pass WF data to services and resources in the operating environment at any time.	–
24	Data Interaction – Environment to Workflow – Pull-Oriented	The ability for a workflow (consists of many workflow process instances of the same kind of workflow) to request workflow level data.	–
25	Data Interaction – Environment to Workflow – Push-Oriented	Services and resources in the operating environment can have the ability to pass workflow-level data so that any WF process instance of that workflow has the ability to use this data.	–
26	Data Interaction – Workflow to Environment – Pull-Oriented	The ability for the workflow engine to handle requests for workflow-level data from services and resources in the operating environment.	–

It is possible to see the benefits of incorporating agents into the blackboard paradigm. Agents can be sent into and out of the workflow engine transporting information freely. For data patterns 15 and 16, a workflow application plug-in program is invoked by specialists (execution of actual WF activities) that can invoke ‘create’ agents to send WF information outside of the workflow engine (data pattern 15) or can invoke a ‘read’ agent to request

information from external entities outside the workflow engine (data pattern 16). For data pattern 17, agents can post data to the blackboard at anytime.

The common description for data pattern 18 is somewhat difficult to interpret when considering the blackboard paradigm. A WF activity or task is executed by a specialist and it does not have the capability to receive and respond to requests for WF data from external entities outside the workflow engine. This pattern should be better understood as when a WF task or activity is active and being executed (specialist is currently engaged), external entities can always request data that is being produced or being used by that WF task or activity by sending a 'read' agent to the blackboard. In any case, all data is readily available to be read by any entity on the blackboard.

Data patterns 19 to 22 deal with data exchange between a WF process instance and entities outside the workflow engine. In this workflow engine, however, a WF process instance is not a distinct explicit entity with any useful or commanding functionality (cannot perform any actions) in the blackboard paradigm. A WF process instance is simply a collection of WF data in a blackboard zone being monitored by specialists. It is modelled implicitly in the blackboard paradigm. Data patterns 19 and 20 describe the ability of a workflow process instance to either request or send data outside of the workflow engine. To accomplish this, specialists can be created that are not directly linked to the actual WF model. They are only responsible for requesting or sending WF data when required. For data patterns 21 and 22, any mention of a WF process instance to describe these patterns is evidently immaterial. 'Create' or 'Update' agents can post data to any zone of the blackboard (workflow process instance) at anytime fulfilling pattern 21 and 'Read' agents can be sent by outside entities to read any data on the blackboard zone thereby fulfilling pattern 22.

Data patterns 23 to 26 are unsupported in this prototype. The lack of support for data pattern 7 (workflow data) has a negative knock-on effect to these set of data patterns. An additional type of blackboard zone to house only workflow data needs to be created for the full support of data pattern 7. This will automatically enable full support for data patterns 23 to 26.

6.3.4 Data Transfer Patterns (27-33)

Data transfer patterns deal with the actual process of how WF data is transferred from one WF component to another. As seen in Table 8, some of these data patterns are supported and some cannot be supported due to the nature of the blackboard.

Communication between specialists is prohibited in the blackboard paradigm. Data that has been generated by a WF activity within one specialist cannot be directly passed to another WF activity residing in another specialist. Therefore, this restriction in the nature of the blackboard system prevents the support of data patterns 27 and 28.

Data pattern 29 (copy in/copy out) is fully supported. Specialists can copy WF variables into temporary variables on the blackboard, operate on them and then overwrite the data from the temporary variable back into the copied variable.

The nature of the blackboard paradigm intrinsically supports data patterns 30 and 31. All WF data that is exchanged between WF components (specialists and agents) can be mutually accessible on the blackboard using a reference (the reference is the name of the WF data). The difference between the two analogous data patterns is that WF data can either be locked (data pattern 31) so WF components have only read-only access or the WF data can be unlocked so that they may be altered by any WF component (data pattern 30). The benefits of splitting the WF data on the blackboard into constants and variables become apparent. To support data pattern 30, specialists can create blackboard variables that can be altered. To support data pattern 31, specialists can create blackboard constants that have only read-only access.

Table 8: Data patterns 27-33 and their support rating.

NO	NAME	DESCRIPTION	SUPPORT
27	Data Transfer by Value – Incoming	Data can be received by any WF component by value thereby not requiring this WF data to have shared names or common address space.	–
28	Data Transfer by Value – Outgoing	Data can be sent by any WF component by value thereby not requiring this WF data to have shared names or common address space.	–
29	Data Transfer – Copy In/Copy Out	WF data can be copied from one address space to another before the start of an execution of a WF activity and resultant value is then copied back during completion.	+
30	Data Transfer by Reference – Unlocked	WF data can be exchanged between WF components by using a reference to a location where that WF data can be reciprocally accessible by all WF components. Concurrency restrictions are not applicable.	+
31	Data Transfer by Reference – Locked	WF data can be exchanged between WF components by using a reference to a location where that WF data can be reciprocally accessible by all WF components. Only read only access is allowed to any WF component to that piece of WF data.	+
32	Data Transformation – Input	A transformation function can be applied to WF data prior to being used by a WF component.	+/–
33	Data Transformation – Output	A transformation function can be applied to WF data prior to being produced by a WF component.	+/–

Data patterns 32 and 33 are partially supported by utilizing data pattern 29 (copy-in/copy-out) to accomplish these two data patterns. There is no mechanism that exists between the blackboard and the specialist that can perform data transformations when blackboard data is being accessed by specialists. Introducing an intermediary between the blackboard and specialist for data transformations will break the blackboard paradigm. Specialists are supposed to be autonomous entities that do not rely on any other entities and can read and post data directly to the blackboard. In order to accomplish these two data patterns, a specialist can pick up the applicable WF variable, perform the transformation (the transformation can be a WF action within the specialist) and then copy the resultant data into the temporary variable (data pattern 32). Once the WF activity has finished using the temporary variable, the specialist can perform the transformation on the temporary variable (another WF action within the specialist) and copy the result back into the initial WF variable (data pattern 33).

6.3.5 Data Routing Patterns (34-40)

The previous data patterns look at WF data separate from the actual execution of WFs. These set of data patterns listed in Table 9 deal with how WF data can be used to directly influence the execution of WFs. WEWE naturally supports these data patterns as specialists constantly react to changes on the blackboard and this can directly influence the operation of WFs.

Table 9: Data patterns 34-40 and their support rating.

NO	NAME	DESCRIPTION	SUPPORT
34	Task Precondition – Data Existence	Preconditions based on data can be specified for WF tasks to check for the existence of WF data prior to the execution of that WF task.	+
35	Task Precondition – Data Value	Preconditions based on data can be specified for WF tasks to check for the value of the WF data prior to the execution of that WF task.	+
36	Task Postcondition – Data Existence	Post-conditions based on data can be specified for WF tasks to check for the existence of WF data during the execution time of that WF task.	+/–
37	Task Postcondition – Data Value	Post-conditions based on data can be specified for WF tasks to check for the value of WF data during the execution time of that WF task.	+/–
38	Event-based Task Trigger	An external event triggers a WF activity.	+
39	Data-based Task Trigger	Execute a WF activity when an expression based on WF data in evaluated to true	+
40	Data-based Routing	The capability of altering the control flow of an active workflow due to evaluation of data –based expressions.	+

The manner in which specialist rules are created intrinsically support data patterns 34 and 35. Code listing 10 displays a few of the many ways specialist rules can be used to check the existence of WF data and their values.

Currently, WEWE is a prototype and only has the ability to create specialist rules following only the condition-action style. Data patterns 36 and 37 require specialist rules to be created in a precondition-action-postcondition style. Furthermore, specialist rules can optionally be created to have only preconditions or only postconditions or both. The introduction of a postcondition to a WF action allows a WF designer to create intrinsic do-while loops for WF actions. As an example, the WF designer can create a WF activity: keep on repeating the 'process fines' action until 'all fines data' is loaded to the blackboard.

```
//For data pattern 34
      If (var.application_submitted != NULL) //check existence
// Another way to accomplish data pattern 34
      If (var.application_submitted) //check existence
//For data pattern 35
      If (var.number_of_applications <= 5) //check value
```

Listing 10: Sample code for specialist rules that support data patterns 34 and 35.

To examine support for data patterns 36 and 37, test specialists with postconditions were created and functionality for evaluating specialist postconditions was inserted into WEWE. No problems were detected while these specialists were active in the workflow engine. However, specialists with postconditions were not used within the WF pattern test scenerios and it is unknown what impact specialists with postconditions will have in the operation of WFs. This needs to be further investigated. For now, it is fair to award partial support to WEWE for data patterns 36 and 37.

Data pattern 38 can be supported through the use of agents. On the occurrence of an external event, an agent can be sent to the blackboard to post a trigger flag. The presence of this trigger flag will prompt some specialist to execute its rules hence executing the required WF activity.

The last two data patterns are suppose to be the toughest patterns for workflow engines to implement [95]. However, the crux of how the blackboard paradigm operates resonates with these two data patterns. A specialist will execute its rules if its conditions are evaluated to true (data pattern 39). The control flow of WFs that guide what path a WF takes is completely dependent on the data present on the blackboard which can be altered by specialists and agents thereby inducing more specialists to execute different WF activities thereby further altering the workflow path (data pattern 40). The core operational aspects of the blackboard paradigm enable the support of these last two data patterns.

6.3.6 Final Results and Comparisons with Existing Workflow Engines

Forty data patterns were examined. If one converts the supports ratings into points as described in Section 6.2.5, a score of 29/40 (72.5%) can be given to WEWE. This score can be boosted to 36/40 (90%) if the following functionality is added:

- The ability to 'spawn off' workflow activities
- Additional WF zones to hold workflow and folder data (data patterns 6 and 7)
- The functionality for specialists to access these new additional WF zones (workflow and folder data blackboard zones).

As part of a future study, test cases need to be created with specialists that have postconditions in their rules and these need to be applied to every WF pattern. Specialist rules with postconditions could possibly simplify the creation of some of the workflow patterns, specifically, the structural and state-based WF patterns.

In 2005, N. Russell, A. H.M. Hofstede, D. Edmond, and W. van der Aalst conducted data pattern tests on the leading WFMSs and at that time found that 48.3% of the data patterns could be implemented by these WFMSs [16]. The same authors have continued this study in [25] and have applied their data pattern tests to new versions of the same workflow engines and many new ones (13 workflow management systems in total) and found that 51.2% of the data patterns can be implemented. A small improvement over time in support can be seen.

The workflow management systems COSA [112] and Oracle BPEL [113] are the top achievers with support levels of 29/40 (72.5%) and 25/40 (62.5%) of data patterns respectively. There are two main reasons why WEWE can support more data patterns than almost all the other workflow engines or management systems that have been investigated by [25]. The first reason is that blackboard paradigm essentially has a global central database: the blackboard. Any component of the workflow can access and alter any kind of data on the blackboard, at anytime. The second and more important reason is the incorporation of software agents into the WF system. Agents allow for increased interoperability. Researchers that are prominent in this field commonly cite poor interoperability and, to a lesser extent, incompatibility with external systems as major limitations in WFMSs [7, 8, 15 and 94]. The benefits of introducing software agents is apparent as no less than eleven data patterns need software agents to be fully supported. If software agents were absent, this workflow engine (WEWE) would have a much lower score (45%) similar to many of the other workflow management systems.

6.4 Summary

This chapter has presented the level of support for data and workflow patterns. It was found that 85% of workflow patterns and 72.5% of data patterns are supported by WEWE. If some of the features that were suggested are

implemented, WEWE will be able to support 95% and 90% of workflow and data patterns respectively. Overall, the data and workflow pattern support is very favourable when compared to existing systems. The next chapter summarizes the work that has been presented and links the findings to the research question.

7 Conclusion

7.1 Introduction

This chapter summarises the findings with regard to implementing a workflow engine using a blackboard approach. A clear answer to the research question proposed in Chapter 4 is given. Ideas and directions for possible future work are given.

7.2 Summary of Work Presented

The concepts behind workflow management and workflow engines have been introduced. Workflow Management Systems (WFMS) consist of many components that allow WF designers to create WFs based on their requirements. These WFs are executed inside a workflow engine residing inside the WFMS. External entities such as software services and humans interact with WFs through WF client applications and invoked applications belonging to this WFMS. This research is only concerned with the core component of the WFMS namely the workflow engine.

The blackboard paradigm was explored in Section 3.2. The blackboard paradigm consists of a blackboard which is a global data structure that is monitored by specialists. The specialists are autonomous components that consist of conditions and actions to perform should the conditions be evaluated to true based on data from the blackboard. Specialists monitor specific blackboard zones searching for an opportunity to make their contribution to the blackboard and execute their actions. The last component in the blackboard system is the controller/orchestrator that can be thought of as a global specialist (monitors the entire blackboard). Depending on the context, it can have many responsibilities such as arranging specialist access to the blackboard, informing specialists of new data being posted to the blackboard and so on. Additionally, another unrelated component that is not a part of the blackboard system is the software agent (see Section 5.4). This component was introduced to allow data from external entities outside the blackboard system to be posted to the blackboard. Data from the blackboard system can be requested from outside entities via these agents.

Many commercially available workflow management systems struggle to support the execution of complex workflows. Furthermore, most of these workflow management systems are not built using any one single architectural pattern but rather have a heterogeneous architecture by amalgamating a group of pure architectural patterns. The opportunistic control innate to the blackboard paradigm can be leveraged to support the execution of complex workflows. The blackboard architecture also seems to support comprehensive workflow functionality. Therefore, it was worthy to investigate whether or not a workflow engine can be built solely utilizing the blackboard paradigm.

Several modelling techniques were mentioned to model workflows in Section 2.3. The fundamental concepts of Petri-nets, which is one of the modelling methods that can model WFs efficiently, can be mapped to the three fundamental entities within a workflow process. These are the process rules, process data and actions to be executed. A process rule can monitor process data. If process data is available for the process rule to be evaluated to true, the process rule can execute an action. This fundamental relationship between these three entities can easily be implemented using blackboard architecture. The process data can be placed on the blackboard and the specialists that monitor the blackboard can contain process rules and actions to be executed if the process rules are evaluated to true.

A workflow engine was built utilizing the blackboard paradigm and compared to traditional workflow engines. A fundamental difference was that most traditional workflow engines are activity based and model WF states implicitly while a workflow engine built on the blackboard paradigm explicitly includes WF states in order to execute WFs.

This research conducted aims to answer the following question:

> *"Can the blackboard paradigm be used to implement a functional and operational workflow engine that can fulfil the requirements set out by the behavioural and informational aspects of workflow management?"*

A workflow engine built on the blackboard paradigm has to be evaluated and compared against other major workflow engines currently in the market. An approach proposed by [88] provides five main aspects of workflow management with quality indicators for each workflow aspect that can be applied to verify their support in a WFMS. This approach was chosen since it can be applied to any WFMS. This empirical approach does not require the conceptual, design and architectural foundations of WFMSs which make it universally applicable to any WFMS. This approach provides a common platform on which a workflow engine built on the paradigm can be judged and can be compared against other workflow engines. These aspects of workflow management were used to formulate the research question stated above.

In order to verify the research question, a prototype was developed using the blackboard paradigm stringently following the guidelines and stipulations provided by [16]. The two main quality indicators are to examine support of the 20 main workflow patterns and 40 main data patterns by creating and executing test cases with these patterns inside the workflow engine.

7.3 Findings

7.3.1 Support for the Behavioural Aspect of WF Engines

The quality indicator for this aspect of workflow management is to examine support for 20 workflow patterns developed by [14, 24 and 25] within WEWE (prototype system). As mentioned in Section 6.2, test cases were built to study support for every single workflow pattern. 85% of the workflow patterns are supported by WEWE.

The final score can be increased from 85% to 95% if the ability to 'spawn off' WF activities is added to this prototype. As mentioned earlier in Section 6.2.2, this can be done easily. Ultimately, the only WF pattern that WEWE cannot support is WF pattern 11 (Implicit Termination). The nature of the blackboard system makes it impossible to support this WF pattern.

The "workflow patterns initiative" have applied their workflows pattern tests to 17 leading workflow engines in 2013 and found that 64.1% of the workflow patterns could be implemented by these workflow engines [25]. Furthermore, it is stated in [111] that many of these workflow management systems were designed and developed to focus on maximum support of these workflow patterns. In stark contrast, the focus of this research was to develop a workflow engine using the blackboard paradigm stringently following the guidelines and stipulations provided by [16]. The maximum support of workflow patterns was never a real consideration in this research. Despite this, a score of 95% (if ability to 'spawn off' WF activities is implemented) is definitely a positive result.

7.3.2 Support for the Informational Aspect of WF Engines

The quality indicator for this aspect of workflow management is to examine support of 'Data patterns' that have been introduced in [15]. The "workflow patterns initiative" have applied their data patterns to 13 popular workflow management systems in 2013 and found that 51.2% of the data patterns can be implemented.

WEWE supports 72.5% of the data patterns. If it can implement the functionality listed in Section 6.3.6, WEWE will be able to support 90% of data patterns. There are two main reasons why WEWE can support more data patterns than any other workflow engine or management system that has been investigated by [25]. The first reason is that the blackboard paradigm essentially has a global central database; the blackboard, in which any component of the blackboard system can access and alter any kind of data on the blackboard. The second reason is the introduction of software agents into the blackboard system to increase interoperability.

If one can implement the functionality listed in Section 6.3.6, WEWE will be able to support 90% of data patterns which is a score greater than most other workflow managements that have been investigated. A score of 90% is definitely a positive result.

7.3.3 Final Finding

The quality indicators and requirements laid out by the functional and operational aspects of the workflow management were implemented during the design and development phase of the prototype. The quality indicators for behavioural and informational aspects of workflow management yielded favourable results. To answer the research question, the blackboard paradigm can be used to implement a functional and operational workflow engine that can fulfil the requirements and pass the quality indicators set out by the behavioural and informational aspects of workflow management.

7.4 Future Work

This research has shown that a blackboard paradigm can be used to build a workflow engine. The next step will be investigating further aspects such as efficiency and performance issues relating to orchestrating workflows using the blackboard paradigm. Workflow engines built on the blackboard paradigm need to spawn multiple threads that run concurrently and this can be resource intensive [114]. Multi-processor computers or even distributed computer environments might be required to run a large number of workflows on a blackboard system. Further research needs to be conducted to determine the level of efficiency of blackboard systems and whether or not they can run workflows on a mass scale.

A common problem that affects all blackboard systems is maintaining data consistency and reducing data contentions of shared data on the blackboard that can concurrently be accessed by multiple specialists. In the context of building workflows engines, this problem can be addressed by employing a blackboard data locking strategy and creating strongly focused specialists that perform a single specific task (discussed in Section 2.10.7). It can be argued that no matter what strategies are taken; memory contention will always be an issue that can hamper performance. Running workflow processes on a massive scale will inevitably lead to memory contentions that will degrade performance. Research needs to be conducted to investigate this issue.

As the situation stands currently, an incomplete WFMS system has been built with WEWE (the core workflow engine) integrated with only a few partially built modules. Further modules of a workflow management system (see Figure 11) need to be developed and this workflow engine needs to be plugged into this WFMS. Before this happens, the limitations of the prototype that have already been mentioned in Sections 6.2.2 and 6.3.6 and the suggestions for improvement have to be implemented. These suggestions will boost the behavioural and informational aspects of the workflow engine.

With the addition of other modules such as WF process definition tools, client applications, user and group collaboration and WF monitoring and administration tools to the WFMS, other aspects of WF management need to be considered. The organizational aspect of WF management needs to be considered when WF client and invoked applications are being designed and developed. Russell, ter Hostede and Edmond have developed 'resource patterns' (like WF and data patterns) as an empirical approach to judge and compare the organizational aspect of workflow management systems against others [115]. Research needs to be conducted as to how the blackboard system can be extended to integrate with these new modules in order to provide maximum support of resource patterns.

When developing the WF monitoring, administration and error handling tools, the failure aspect of workflow management needs to be considered. Error and exception handling patterns have been developed as well to determine the level of error and exception handling support that exists within any WFMS [116]. Research needs to be conducted to verify the level of support for error and exception handling patterns the workflow engine can provide. The workflow engine might need to be extended to accommodate some of these patterns but the guidelines and stipulations laid out by [58] should still be followed so that the workflow engine still conforms to the blackboard paradigm.

It is suggested in Section 5.6 that the current WF definition interface be discarded and a more user friendly and intuitive interface should be built that abstracts all implementation details of the workflow engine. Presentation patterns have been created by [117] and [118] which need to be considered when developing this user friendly WF definition interface. As more and more modules of the WFMS are being designed and developed, further aspects of workflow management such as security, causality, integrity, failure, history, recovery, quality and autonomy need to be considered.

At present, this research has shown that a blackboard paradigm can be utilized to develop a workflow engine. Furthermore, the research has shown that a workflow engine built utilizing the blackboard paradigm can outperform many open-source and commercial WFMSs (collection of WFMSs found in [14], [15] and [25]) in behavioural and informational aspects. The research can now be expanded to other components of the workflow management system and how the workflow engine (central component) interacts with these new components. It is conceivable that a fully fledged WFMS with a workflow engine at its centre, built using blackboard architecture would offer significant advantages over traditional WFMSs.

8 References

[1] G. Kappel, S. Rausch-Schott, and W. Retschitzegger. A framework for workflow management systems based on objects, rules and roles. *ACM Computing Surveys (CSUR) 32*, no. 1es, 2000.

[2] Glossary, Terminology and Glossary, 3rd Edition. Document No WFMC-TC-1011. *Workflow Management Coalition*. Winchester, 1999.

[3] F. Leymann and D. Roller. *Production workflow: Concepts and Techniques*. Prentice-Hall, 2000.

[4] M. S. Puccini. *Executable Models for Extensible Workflow Engines*. PhD diss., UNIVERSIDAD DE LOS ANDES, February, 2011.

[5] OASIS Web Services Business Process Execution Language (WSBPEL) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, Last Accessed 15 January, 2015.

[6] Yet Another Workflow Language (YAWL). http://www.yawlfoundation.org/, Last Accessed 15 January, 2015.

[7] D. Georgakopoulos, M. Hornick and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases 3*, no. 2 199: pp 119-153, 1999.

[8] A. El Abbadi, D. Agrawal, G. Alonso and C. Mohan. Functionality and limitations of current workflow management systems. http://www.almaden.ibm.com/cs/exotica/wfmsys.ps. Last Accessed 15 January, 2015.

[9] P. Heinl, S. Horn, S. Jablonski, J. Neeb, K. Stein and M. Teschke. A comprehensive approach to flexibility in workflow management systems. *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 2, pp. 79-88. ACM, 1999.

[10] E. Deelman, D. Gannon, M. Shields and I. Taylor. Workflows for e-Science. *Scientific Workflows for Grids*, Springer Publishing Company, Incorporated, 2014.

[11] S. Bowers, T. McPhillips, B. Ludscher and M. Weske. Scientific Workflows: Business as Usual? *Business Process Management*, volume 5701/2009 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009.

[12] S. Christensen , B. Jonathan, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, K. Van Hee and M. Weber. *The Petri net Markup Language: Concepts, Technology, and Tools*. Springer Berlin Heidelberg, 2003.

[13] T. József. P-graph-based workflow modelling. *Acta Polytechnica Hungarica 4*, no. 1 : pp 75-88, 2007.

[14] W. van Der Aalst, A. H.M. Hofstede, B. Kiepuszewski and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases 14 (1)*, pp. 5-51. Doi:10.1023/A:1022883727209, 2003.

[15] N. Russell, A. H.M. Hofstede, D. Edmond, and W. van der Aalst. Workflow data patterns: Identification, representation and tool support. *Conceptual Modeling–ER 2005*, pp 353-368, 2005.

[16] B. Hayes-Roth. The blackboard architecture: A general framework for problem solving?. Heuristic Programming Project, Computer Science Department, Stanford University, 1983.

[17] D. Hollingsworth and U. K. Hampshire. Workflow management coalition the workflow reference model. *Workflow Management Coalition*, 68, 1993.

[18] S. Joosten. Trigger Modelling for Workflow Analysis. *Proc. CON '94: Workflow Management*, pp. 236-247, publ: R. Oldenbourg, Vienna, Munchen, ISBN 3-7029-0397-6, 1994.

[19] G. Weikum and D. Wodtke. A formal foundation for distributed workflow execution based on state charts. *In Database Theory—ICDT'97*, pp. 230-246. Springer Berlin Heidelberg, 1997.

[20] A. H.M. Hofstede and D. Marlon. UML activity diagrams as a workflow specification language. *UML 2001—The Unified Modelling Language. Modelling Languages, Concepts, and Tools*, pp. 76-90. Springer Berlin Heidelberg, 2001.

[21] W. Van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers 8*, no. 01, pp 21-66, 1998.

[22] S. Handley. On the use of a directed acyclic graph to represent a population of computer programs. *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence, Proceedings of the First IEEE Conference on*, pp. 154-159. IEEE, 1994.

[23] J. Ahrens, J. Freire, L. Lins, E. Santos and C.T. Silva. A first study on clustering collections of workflow graphs. *Provenance and Annotation of Data and Processes*, pp. 160-173. Springer Berlin Heidelberg, 2008.

[24] A. H.M. Hofstede, N. Mulyar and N. Russell. Workflow control flow patterns: A revised view. *The Workflow Patterns Initiative*, 2006.

[25] Workflow Patterns Initiative. http://www.workflowpatterns.com, Last Accessed 15 January 2015.

[26] Java Workflow Tooling (JWT). http://www.eclipse.org/proposals/jwt/, Last Accessed 15 January, 2015.

[27] D. Hollinsworth. The Workflow Reference Model. *Workflow Management Coalition*, TC00-1003, 1994.

[28] W. Van der Aalst and P.J.S. Berens. Beyond workflow management: product-driven case handling. *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pp. 42-51. ACM, 2001.

[29] Bergmann. Design and Implementation of a Workflow Engine. Diploma Thesis, University of Bonn, Germany, August, 2009.

[30] SOA Product Review — ActiveBPEL 3.0 from Active Endpoints. Maurer, P. http://soa.sys-con.com/node/318429, Last Accessed 15 January 2015.

[31] Activiti Vision. http://www.activiti.org/vision.html, Last Accessed 15 January 2015.

[32] Drools 5 and jBPM5 Architectural Overview. http://bpmgeek.com/blog/drools-5-and-jbpm5-architectural-overview, Last Accessed 15 January 2015.

[33] J. Cachopo , S.M. Fernandes and A. R. Silva. Supporting evolution in workflow definition languages. *SOFSEM 2004: Theory and Practice of Computer Science*, pp. 208-217. Springer Berlin Heidelberg, 2004.

[34] Flowmark, I. B. M. Modeling Workflow. IBM Corporation, September 1994.

[35] S. Breutel, M. Dumas, A. H.M. Hofstede, C. Ouyang, E. Verbeek and W. Van Der Aalst. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming 67*, no. 2: pp 162-198, 2007.

[36] G. Vossen and M. Wesk. Workflow Languages. *Handbook on Architectures of Information Systems*, pp. 359-379. Springer Berlin Heidelberg, 1998.

[37] P. Athena. Case handling with flower: Beyond workflow. *Pallas Athena BV, Apeldoorn, The Netherlands 11*, 2002.

[38] H. Gaur. BPEL Cookbook: Best Practices for SOA-based Integration and Composite Applications Development; Ten Practical Real-world Case Studies Combining Business Process Management and Web Services Orchestration. Packt Publishing Ltd, 2006.

[39] G. Hackmann, M. Haitjema, C. Gill and G. Roman. Sliver: A BPEL workflow process execution engine for mobile devices. *Service-Oriented Computing–ICSOC 2006*, pp. 503-508. Springer Berlin Heidelberg, 2006.

[40] S. Moser and T. van Lessen. Developing, deploying and running a hello world BPEL process with the Eclipse BPEL designer and Apache ODE, 2011.

[41] A. Faderman, P. Koletzke and P. Dorsey. Oracle JDeveloper 10g Handbook. McGraw-Hill, Inc., 2004.

[42] T. Gunarathne, D. Premalal, T. Wijethilake, I. Kumara and A. Kumar. BPEL-Mora: lightweight embeddable extensible BPEL engine. *Emerging Web Services Technology*, pp. 3-20. Birkhäuser Basel, 2007.

[43] ActiveBPEL engine architecture. http://www.activebpel.org/docs/architecture.html, Last Accessed 15 January, 2015.

[44] O. Coupelon. Analyse et optimisation de l'architecture des moteurs BPEL. Rapport de stage, Master Recherche Informatique, Université Blaise Pascal, 2007.

[45] Workflow Pattern 8 (Multi-Merge). http://www.workflowpatterns.com/patterns/control/advanced_ branching/wcp8.php, Last Accessed 15 January 2015.

[46] Workflow Pattern 10 (Arbitrary Cycles). http://www.workflowpatterns.com/patterns/control/structural /wcp10.php, Last Accessed 16 December 2014.

[47] YAWL: Official Website. http://yawlfoundation.org/, Last Accessed 17 December 2014.

[48] W. Van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers 8*, no. 01 (1998): pp 21-66, 1998.

[49] W. Van Der Aalst, L. Aldred, M. Dumas and A. H.M. Hofstede. Design and implementation of the YAWL system. *Advanced Information Systems Engineering*, pp. 142-159. Springer Berlin Heidelberg, 2004.

[50] W. Van der Aalst and A. H.M. Hofstede. YAWL: yet another workflow language. *Information systems 30*, no. 4 (2005): pp 245-275, 2005.

[51] YAWL: Technical Manual. http://www.yawlfoundation.org/manuals/YAWLTechnicalManual2.1.pdf, Last Accessed 17 December 2014.

[52] L. Pomello, S. Haddad. Application and Theory of Petri Nets. *33rd International Conference*, PETRI NETS 2012 Hamburg, Germany, June 25-29, 2012.

[53] Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy and Y. Liu. Grid-Flow: a Grid-enabled scientific workflow system with a Petri-net-based interface. *Concurrency and Computation: Practice and Experience 18*, no. 10 (2006): pp 1115-1140, 2006.

[54] K. Maheshwari, and J. Montagnat. Workflow development using both visual and script-based representation. *Services (SERVICES-1), 2010 6th World Congress on*, pp. 328-335. IEEE, 2010.

[55] S. Jablonski and C. Bussler, Workflow Management: Modeling Concepts, Architecture, and Implementation, *International Thomson Computer Press*, 1996.

[56] GWENDIA Workflow Language Proposal. https://gwendia.i3s.unice.fr/lib/exe/fetch.php?media=public _namespace:l1.2.pdf, Last Accessed 21 December 2014.

[57] B. Buchanan and R. G. Smith. Fundamentals of expert systems. *Annual Review of Computer Science 3*, no. 1 (1988): 23-58, 1988.

[58] H.P. Nii. The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *Association for the Advancement of Artificial Intelligence*, Vol 7 No. 2, 2002.

[59] D. Corkill, K. Gallagher and K. E. Murray. GBB: A generic blackboard development system. *Proceedings of the National Conference on Artificial Intelligence*, pp 1008–1014, Philadelphia, Pennsylvania, August 1986.

[60] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett and A. Seiver. Intelligent monitoring and control. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp 243–249, Detroit, Michigan, August 1989.

[61] R. Englemore and T. Morgan, Blackboard Systems. Pub. Addison-Wesley Pub. Co. 1988.

[62] B. Hayes-Roth and H. Hewett. Learning control heuristics in a blackboard environment. Technical Report HPP-85-2, Computer Science Department, Stanford University, Palo Alto, CA, 1985.

[63] N. Carver and V. Lesser. Evolution of blackboard control architectures. *Expert systems with applications 7*, no. 1 (1994): pp 1-30, 1994.

[64] D. Corkill. Advanced architecture: Concurrency and parallelism. *V. Jagannathan, R. Dodhiawala, & L Baum (Eds.), Blackboard Architectures and Applications*, pp. 77-83, New York: Academic Press, 1989.

[65] B. Hayes-Roth. Blackboard architecture for control. *Journal of Artificial Intelligence*, 26, pp 251-321, 1985.

[66] H.P. Nii, E. Feigenbaum, J. Anton and A.J. Rockmore. Signal-to-symbol transformation: HASP/SIAP case study. *The AI Magazine*, 3, 23-35, 1982.

[67] B. Hayes-Roth and M. Hewett. BBl: An implementation of the blackboard control architecture. *R. Engelmore & T. Morgan (Eds.), Blackboard systems*, pp. 297-313, MA: Addison-Wesley, 1988.

[68] B. Hayes-Roth, M.V. Johnson, A. Garvey and R.M. Hewe. Application of the BBI blackboard architecture to arrangement assembly tasks. *International Journal for Artificial Intelligence in Engineering*, 1, pp 85-94, 1986.

[69] H.P. Nii. Blackboard Systems at the Architecture Level. *Expert Systems with Applications*, Vol. 7, pp. 43-54, 1994.

[70] B.M. Michelson. Event-driven architecture overview. Patricia Seybold Group 2, 2006.

[71] A.A. Hopgood, H. J. Phillips, P. D. Picton and N. J. Braithwaite. Fuzzy logic in a blackboard system for controlling plasma deposition processes. *Artificial Intelligence in Engineering 12*, no. 3 (1998): pp 253-260, 1988.

[72] S. E. Lander, V. R. Lesser and M. E. Connell. Knowledge-based conflict resolution for cooperation among expert agents. *D. Sriram, R. Logher, and S. Fukuda, editors, Computer-Aided Cooperative Product Development*, pp 183–198, Springer Verlag, 1991.

[73] H.P. Nii. Blackboard Systems: The Blackboard Model of Problem–solving and the Evolution of Blackboard Architectures. *AI Magazine, 7 (2)*, pp. 38 – 53, 1986.

[74] D. Corkill. Advanced Architectures: Concurrency and Parallelism. *Blackboard Architectures and Applications*, ed. V . Jagannathan et al., Academic Press, Inc., 1989.

[75] J. McManus. A Parallel Distributed System for Aircraft Tactical Decision Generation. *Proceedings of the 9th Digital Avionics Systems Conference*, pp.505 – 512, 1990.

[76] R. Hueschen and J. McManus. Application of AI Methods to Aircraft Guidance and Control. *Proceedings 1988 American Control Conference*, June 15-17, pp. 195 – 201, 1988.

[77] D. D. Corkill. Design alternatives for parallel and distributed blackboard systems. *Blackboard Architectures and Applications*, V. Jagannathan, R. Dodhiawala, and L. S. Baum, Eds. San Diego, CA: Academic, pp. 99-136, 1989.

[78] H. F. Korth. Locking primitives in a database system. *Journal of the ACM (JACM) 30*, no. 1, pp 55-79, 1983.

[79] J. McManus. A Parallel Distributed System for Aircraft Tactical Decision Generation. *Proceedings of the 9th Digital Avionics Systems Conference*, pp.505 – 512, 1990.

[80] R. M. Hueschen and J. McManus. Application of AI Methods to Aircraft Guidance and Control. *Proceedings 1988 American Control Conference*, June 15-17, pp. 195 – 201, 1988.

[81] J. McManus and K. H. Goodrich. Application of Artificial Intelligence (AI) Programming Techniques to Tactical Guidance for Fighter Aircraft. *Proceedings 1989 AIAA Guidance, Navigation, and Control Conference*, AIAA Paper # 89-3525, pp. 851-858, 1989.

[82] L. D. Erman, F. Hayes-Roth. The Hearsay-II Speech Understanding System: Integrating knowledge to resolve uncertainty. *ACM Computing Survey*, 12, pp 213-253, 1980.

[83] E. Schikuta and K. Kofler. Using blackboard system to automate and optimize workflow orchestrations. *Emerging Technologies, 2009*. ICET 2009. International Conference on, pp. 173-178. IEEE, 2009.

[84] S. K. Stegemann, B. Funk and T. Slotos. A blackboard architecture for workflows. *CAiSE Forum*, vol. 247. 2007.

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

[86] A. P. Sheth, W. M. P. Van der Aalst and I. B. Arpinar. Processes driving the networked economy. *IEEE Concurrency*, 7(3):pp 18–31, 1999.

[87] S. Jablonski, and C. Bussler. Workflow management: modeling concepts, architecture and implementation. 1996.

[88] S. Petkov, E. Oren and A. Haller. Aspects in workflow management. *Galway, Ireland: DERI, Digital Enterprise Research Institute*, 2005.

[89] M. Rusinkiewicz and A. Sheth, Specification and Execution of Transactional Workflows. *Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim, Ed., ACM Press, 1994.

[90] M. Dumas and A. Hofstede. UML activity diagrams as a workflow specification language. *UML 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pp 76-90, 2001.

[91] W. Van der Aalst. The application of Petri nets to workflow management. *Journal of circuits, systems, and computers 8*, 01: pp 21-66, 1998.

[92] M. Dumas and A. Hofstede. UML Activity Diagrams as a Workflow Specification Language. *Fourth International Conference on the Unified Modelling Language (UML 2001)*, pages 76–90, Toronto, Canada, 2001.

[93] D. Riehle and H. Züllighoven. Understanding and using patterns in software development. *TAPOS 2*, no. 1 (1996): pp 3-13, 1996.

[94] P.M. Steele and A.B. Zaslavsky. The Role of Meta Models in Federating System Modelling Techniques. *Proceedings of the 12th International Conference on the Entity-Relationship Approach – ER `93*, Eds.: R. A. Elmasri, V. Koura-majian, B. Thalheim. Berlin et al., pp. 315-326, 1994.

[95] G. Gottlob, M. Schrefl and B. Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems (TOIS) 14*, no. 3: pp 268-296, 1996.

[96] Concurrent HashMap (Java Platform SE 7). http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html, Last Accessed 6 January, 2014.

[97] D. Gawlick and S. Mishra. Information sharing with the Oracle database. *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems*, pp. 1-6. ACM, 2003.

[98] F. Hayes-Roth. Rule-based systems. *Communications of the ACM 28*, no. 9 (1985):pp 921-932, 1985.

[99] Difference between asynchronous and synchronous events. http://www.slideshare.net/umarali1981/difference-between-asynchronous-event-handler-and-synchronous-event-handler-in-sharepoint, Last Accessed 19 December 2014.

[100] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM 37*, no. 7 (1994): pp 48-53, 1994.

[101] T. Selker. A Teaching Agent that learns. *Communications of the ACM 37*, pp 92-99, 1994.

[102] R. Gopalapuram. Message propagation from Oracle to WebSphere MQ using messaging gateway. PhD diss., California State University, Sacramento, 2010.

[103] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *Intelligent Agents III Agent Theories, Architectures, and Languages*, pp 21-35, 1997.

[104] Spring Framework. http://www.springsource.org/spring-framework, Last Accessed December 25, 2014.

[105] Spring Security. http://static.springsource.org/spring-security/site/, Last Accessed December 25, 2014.

[106] H. Velthuijsen. *The Nature and Applicability of the Blackboard Architecture*. PTT research, 1992.

[107] G. T. Byrd and M. J. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE 87*, no. 3: pp 456-466, 1999.

[108] LinkedBlockingQueue. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html, Last Accessed 27 December 2014.

[109] CountDownLatch. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html, Last Accessed 27 December 2014.

[110] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. *ACM Sigplan Notices*, vol. 37, no. 11, pp. 161-173. ACM, 2002.

[111] Workflow Patterns Initiative - Evaluations. http://www.workflowpatterns.com/impact/evaluations.php, Last Accessed 29 December 2014.

[112] COSA Pattern Evaluation Results. http://www.workflowpatterns.com/evaluations/commercial/cosa.php, Last Accessed 28 December 2014.

[113] Oracle BPEL Pattern Evaluation Results. http://www.workflowpatterns.com/evaluations/standard/oraclebpel.php, Last Accessed 28 December 2014.

[114] V.C. Hamacher, Z.G. Vranesic and S.G. Zaky, Computer Organization, McGraw Hill, Singapore, 1996.

[115] A. Hofstede, D. Edmond and W. MP van der Aalst. Workflow resource patterns. Beta, Research School for Operations Management and Logistics, 2005.

[116] N. Russell, W. van der Aalst and A. ter Hofstede. Workflow exception patterns. *Advanced Information Systems Engineering*, pp. 288-302. Springer Berlin Heidelberg, 2006.

[117] M. La Rosa, A. Hofstede, P. Wohed, H. Reijers, J. Mendling and W. van der Aalst. Managing process model complexity via concrete syntax modifications. *Industrial Informatics, IEEE Transactions on 7*, no. 2: pp 255-265, 2011.

[118] M. La Rosa, P. Wohed, J. Mendling, A. Hofstede, H. Reijers and W. van der Aalst. Managing process model complexity via abstract syntax modifications. *Industrial Informatics, IEEE Transactions on 7*, no. 4: pp 614-629, 2011.

9 Bibliography

G. Alonso, D. Agrawal, A. El Abbadi and C. Mohan. Functionality and limitations of current workflow management systems. *IEEE Expert 12*, no. 5: pp 105-111, 2011.

T. A. Barrass, Y. Wu, D. Semeniouk, D. Bonacorsi, L. Newbold and T. Tuura. Wildish et al. Software agents in data and workflow management. CERN-CMS-CR-2004-053, 2004.

F. Bellifemine, G. Caire and D. Greenwood. Developing multi-agent systems with JADE. Vol. 7. John Wiley & Sons, 2007.

F. Bellifemine, F. Bergenti, G. Caire and A. Poggi. JADE—a java agent development framework. *Multi-Agent Programming*, pp. 125-147. Springer US, 2005.

A. Bonifati, F. Casati, U. Dayal and M. Shan. Warehousing workflow data: Challenges and opportunities. *VLDB, vol. 1*, pp. 649-652, 2001.

E. Börger. Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL. *Software & Systems Modeling 11*, no. 3: pp 305-318, 2012.

B. Brzykcy, J. Martinek, A. Meissner and P. Skrzypczynski. Multi-agent blackboard architecture for a mobile robot. *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 4, pp. 2369-2374. IEEE, 2001.

R. Dijkman, M. Dumas and C. Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology 50*, no. 12: pp 1281-1294, 2002.

T. Dörnemann, T. Friese, S. Herdt, E. Juhnke and B. Freisleben. Grid workflow modelling using grid-specific BPEL extensions. *Proceedings of German e-Science Conference*, vol. 2007, pp. 1-9. 2007.

R. Eshuis and R. Wieringa. Comparing Petri net and activity diagram variants for workflow modelling–a quest for reactive Petri nets. *Petri Net Technology for Communication-Based Systems*, pp. 321-351. Springer Berlin Heidelberg, 2003.

C. Hagen and G. Alonso. Exception handling in workflow management systems. *Software Engineering, IEEE Transactions on 26*, no. 10: pp 943-958, 2010.

S. Krishnan, P. Wagstrom and G. Von Laszewski. GSFL: A workflow framework for grid services. Preprint ANL/MCS-P980-0802, Argonne National Laboratory 9700, 2002.

Y. Lei and M. P. Singh. A comparison of workflow metamodels. *Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling at ER*, vol. 97, 1997.

V. Lesser, B. Horling, F. Klassner, A. Raja, T. Wagner and S. XQ Zhang. BIG: An agent for resource-bounded information gathering and decision making. *Artificial Intelligence 118*, no. 1: pp 197-244, 2000.

S. Ling and H. Schmidt. Time Petri nets for workflow modelling and analysis. *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 4, pp. 3039-3044, IEEE, 2000.

J. Mendling, G. Neumann and M. Nüttgens. Towards workflow pattern support of event-driven process chains (EPC). *In Proc. of the 2nd Workshop XML4BPM*, vol. 2005, pp. 23-38, 2005.

S. Pandey, D. Karunamoorthy and R. Buyya. Workflow engine for clouds. *Cloud Computing: Principles and Paradigms*, pp 321-342, 2011.

N. Rodrigues, P. Monteiro Jr, O. Sampaio, J. de Souza and G. Zimbrão. Autonomic business processes scalable architecture. *Business Process Management Workshops*, pp. 78-83. Springer Berlin Heidelberg, 2008.

S. Sadiq, M. Orlowska, W. Sadiq and C. Foulger. Data flow and validation in workflow modelling. *Proceedings of the 15th Australasian database conference-Volume 27*, pp. 207-214. Australian Computer Society, Inc., 2004.

G. Shegalov, M. Gillmann and G. Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *The VLDB Journal—The International Journal on Very Large Data Bases 10*, no. 1 (2001): p 91-103, 2001.

A. Sheth. From contemporary workflow process automation to adaptive and dynamic work activity coordination and collaboration. *In Database and Expert Systems Applications, 1997. Proceedings., Eighth International Workshop on*, pp. 24-27. IEEE, 1997.

G. Singh, K. Vahi, A. Ramakrishnan, G. Mehta, E. Deelman, H. Zhao, R. Sakellariou et al. Optimizing workflow data footprint. *Scientific Programming 15*, no. 4: pp 249-268, 2007.

H. Smith and P. Fingar. Workflow is just a Pi process. *BPTrends*, pp 1-36, 2006.

W. Van Der Aalst and K. Van Hee. Workflow management: models, methods, and systems. MIT press, 2004.

W. Van Der Aalst. Three good reasons for using a Petri-net-based workflow management system. *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)*, pp. 179-201. 1996.

H. Verbeek and W. van der Aalst. Analyzing BPEL processes using Petri nets. *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pp. 59-78. 2005.

H. Zhang, R. Yin and S. Zhang. Design of Workflow Engine Based on Web. *Computer Engineering 4*, pp33-45, 2004.

Remoting and web services using Spring. http://docs.spring.io/spring/docs/3.0.x/spring-frameworkreference/ html/remoting.html, Last Accessed 27 December, 2014.

Spring Web MVC framework. http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html, Last Accessed 15 January 2015.

Remote Method Invocation. http://docs.oracle.com/javase/tutorial/rmi/, Last Accessed 19 January 2015.

Java Annotations. http://docs.oracle.com/javase/tutorial/java/annotations/, Last Accessed 25 January 2015.

Java Reflection. http://docs.oracle.com/javase/tutorial/reflect/, Last Accessed 25 January 2015.

Appendices

A. Blackboard Implementation

A.1 Introduction

The implementation details of the blackboard component of the workflow engine are provided. This specifically includes the UML diagrams and database schemas to provide a concise picture of how the blackboard component is built.

A.2 Blackboard Structure

In Section 5.2, it was stated that there exists a one-to-many relationship between a workflow and workflow process instances. This fundamental relationship in WFMSs shown in Figure A.1 forms the basis on which the blackboard's structure is designed.

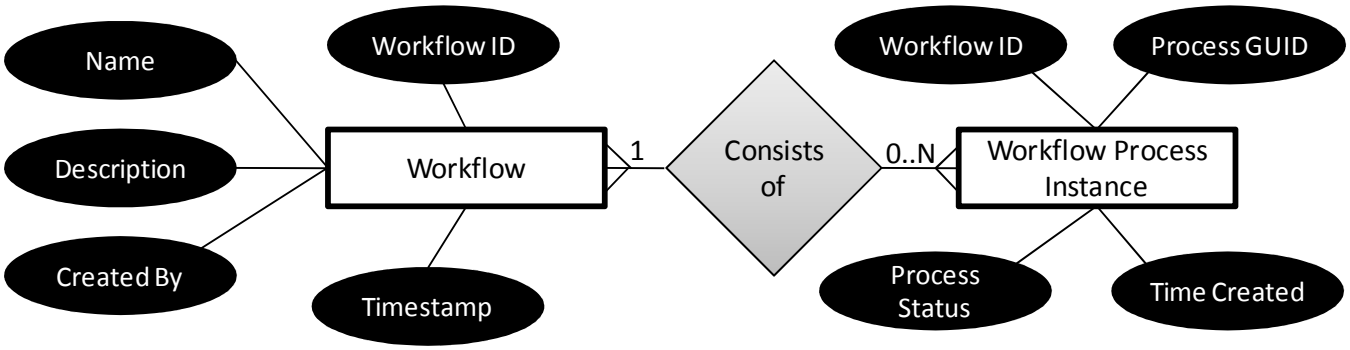

Figure A.1: The Entity Relationship Diagram (ERD) of the fundamental relationship between workflows and process instances.

The relationship in Figure A.1 assists to form objects that can be organized hierarchically. Firstly, a 'workflow process instance' object is created that can contain all process and control data of a single process instance. Functionality that can be used by specialists and the controller/orchestrator for retrieving information (by reference or by value), editing, deleting and adding data is also added to this object. Thereafter, a 'workflow' object is created that can simply contain all the workflow process instance objects. The workflow object is used for grouping workflow process instances of the same type. These workflow objects containing their WF process instances can be placed on the blackboard. Thus, a simple and intuitive object hierarchy of two levels is used to structure the blackboard shown in Figure A.2.

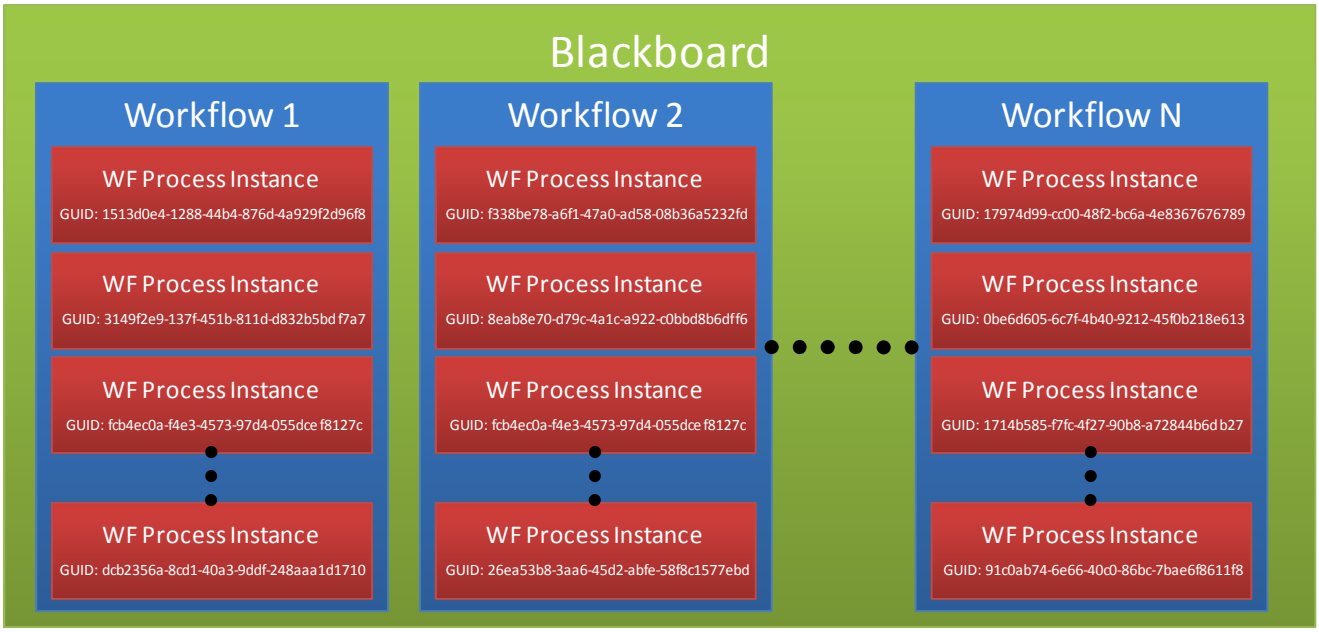

Figure A.2: The structure of WEWE's blackboard with implementation details.

It can be seen from Figure A.2 that the blackboard is split into zones for each workflow process instance. This is beneficial as a set of specialists for a specific workflow process instance can specifically monitor information pertinent to that WF process instance instead of arbitrarily scanning large irrelevant portions of the blackboard to find information of relevance. Since WFMSs could possibly have many other applications and databases connected to it, WF process instances are assigned Global Unique Identifiers (GUIDs) to ensure data integrity.

A blackboard of this nature with some sort of locking strategy is easily implementable using Java's map data structures specifically the concurrent hash map data structure [1]. The blackboard was designed to essentially consist of a concurrent hash map of workflow objects and these workflow objects further consist of a concurrent hash map of WF process instances as shown in Figure A.3. Specialists that want to monitor and interact with a particular workflow process instance (a zone) can hash the blackboard by using the ID of the workflow and the GUID of the WF process instance to retrieve the required WF process instance object. If a new WF process is

spawned, the orchestrator/controller will create it inside the staging zone and prepare the object before transferring it to its workflow zone.

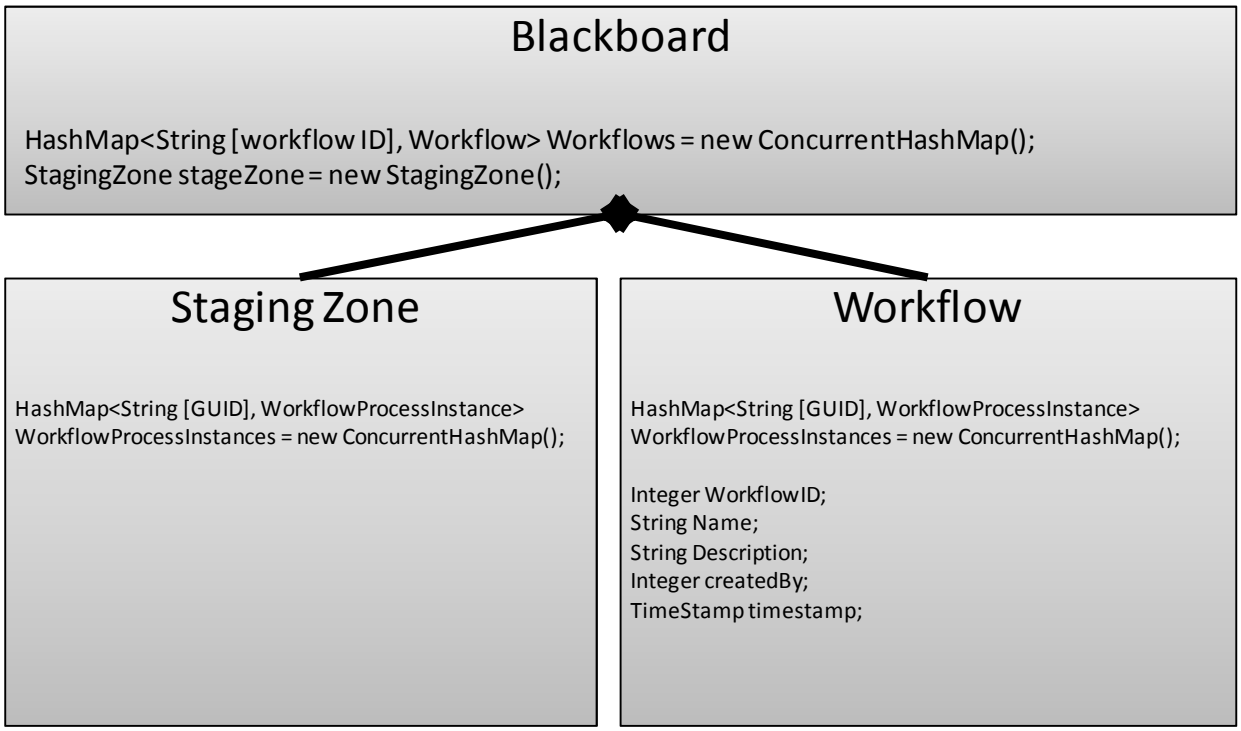

Figure A.3: UML diagram of the blackboard structure with implementation details.

A.3 Blackboard Database Schema

A database design is created that enables full auditing similar to the database design implemented in date tracking systems for oracle databases [2]. The next few paragraphs discuss the database design structure in more detail to accommodate full data auditing.

Table A.1: The schema for the table wewe_workflows.

Table "wewe_workflows"		
field	type(size)	default/options/indexes/constraints
workflow_id	serial	auto incrementing, primary key, not null
title	varchar(250)	not null
short_description	varchar(500)	not null
description	text	
timestamp	timestamp	currentdatetime + timezone
created_by	varchar(50)	not null default "wewe"
workflow_status	integer	default

When a workflow designer creates a new workflow, Table A.1 is used to save its metadata. The workflow ID is used as a foreign key for all other tables. All information under the umbrella of this workflow (WF process instances, specialists, forms information etc) will be somehow linked to this workflow ID. Table A.2 records all the workflow process instance objects.

Table A.2: The schema for the table wewe_process_instances.

Table "wewe_process_instances"		
field	type(size)	default/options/indexes/constraints
guid	guid	primary key, not null
workflow_id	integer	foreign key (wewe_workflows:workflow_id)
timestamp	timestamp	currentdatetime + timezone
process_status	integer	not null

When a new workflow process instance object is created on the blackboard, the object itself inserts a record into Table A.2. Once the record is inserted, the object will retrieve its GUID from this newly inserted record. This table also records the status of each workflow process instance as an integer value. Table A.3 lists the meanings of each integer value.

Table A.3: A list of all workflow process instance statuses and their descriptions.

Status Number	*Meaning*	*Description*
1	Active	WF process instance is running tasks and activities (Specialist is currently interacting)
2	Idle	WF process instance is running but no tasks and activities are currently being executed (No specialists interaction)
3	Completed	WF process instance has been completed
4	Paused	WF process instance has been paused by user
5	Inactive (new)	WF process instance is in the process of being setup on the blackboard
6	Inactive	WF process instance has been cancelled by user
7	Halted due to error	WF process instance has been paused due to an error
8	Cancelled	WF process instance has been cancelled.

The status of a workflow process instance can change depending on the circumstances. The workflow instance object immediately updates its status in Table A.2 when a change in status occurs.

WF process constants are created either when a WF process instance is spawned or when specialists add constants into a WF process instance. When a constant is added to a WF process instance object, it immediately saves the details of that constant to Table A.4.

Table A.4: The schema for the table wewe_wf_process_constants.

Table "wewe_wf_process_constants"		
field	type(size)	default/options/indexes/constraints
const_id	serial	auto increment, primary key, not null
workflow_id	integer	foreign key (wewe_workflows:workflow_id)
guid	guid	foreign key (wewe_process_instances:guid)
constant_name	varchar(100)	not null constraint
constant_value	text	not null constraint
constant_type	varchar(100)	not null constraint

Once a constant has been created in a workflow process instance object, it cannot be changed or can be removed and will be perennial until the completion of the workflow process or when a workflow process instance is removed from the blackboard. So no auditing of constants is required. Only a simple constraint is added that restricts more than one constant of the same name to be created within a workflow process instance.

WF process variables can be added, altered and removed by specialists by extension WF tasks and activities at will and at anytime within the lifetime of a workflow process instance, and by doing so, advance the WF process's path within the boundaries of the workflow. All changes to variables must be recorded for auditing purposes. Two tables with their schema described in Tables A.5 and A.6 are used track changes to variables.

Table A.5: The schema for the table wewe_wf_process_iterations.

Table "wewe_wf_process_iterations"		
field	type(size)	default/options/indexes/constraints
iteration_id	serial	auto increment, Primary Key, Not Null
specialist	varchar(100)	NOT NULL constraint
timestamp	timestamp	currentdatetime + timezone
workflow_id	int	Foreign Key (wewe_workflows:workflow_id)
guid	guid	Foreign Key (wewe_process_instances:guid)

Table A.6: The schema for the table wewe_wf_process_variables.

Table "wewe_wf_process_variables"		
field	type(size)	default/options/indexes/constraints
var_id	serial	auto increment, primary key, not null
workflow_id	integer	foreign key (wewe_workflows:workflow_id)
guid	guid	foreign key (wewe_process_instances:guid)
iteration_id	integer	foreign key (wewe_wf_process_iterations:iteration_id)
var_name	varchar(100)	not null constraint

var_value	text	no constraints or defaults
var_type	varchar(100)	not null constraint
timestamp	timestamp	currentdatetime + timezone
modified_by	varchar(100)	not null constraint
is_active	small integer	not null constraint

Table A.5 records all the interactions between specialists and the workflow process instance object whenever a variable is created, altered or removed. Each specialist interaction can be termed as a workflow process iteration since a specialist interacting with the WF process object generally means that the WF process instance has advanced (actions are being performed) within the variables of the workflow definition. Every time a variable is created, altered or removed, the workflow process instance inserts a record in Table A.6 with a reference of the iteration (from Table A.5) of the workflow process instance. New iteration and variable records are added when a variable is altered. If a variable is removed from a workflow process instance, the 'is_active' field is updated to false instead of removing records pertaining to that variable.

In the event of a system reload, WEWE is able to use the data stored into these set of tables to completely recreate the blackboard and revive all uncompleted workflows at their point of stoppage. All workflow process iterations are stored in the logs component of the workflow instance. This can be used by administrators who want to rewind or fast forward workflow processes to any workflow state.

A.4 Data Types

Constants and variables that are saved in the database lose their data type since their string value is saved into the database. Therefore, it is necessary to have a type field (var_type and constant_type) to also save the data type of WF variables and constants. A 'data type manager' exists within WEWE which takes WF data (WF variables and constants) with their type and 'stringify's' them so that they can be saved into the database. Conversely, variables and constants already saved in the database can be retrieved by WF process instances using their string value and data type that can be parsed by the data type manager. The workflow process instance objects use the data type manager as a translation layer between them and the database and shown in Figure A.4.

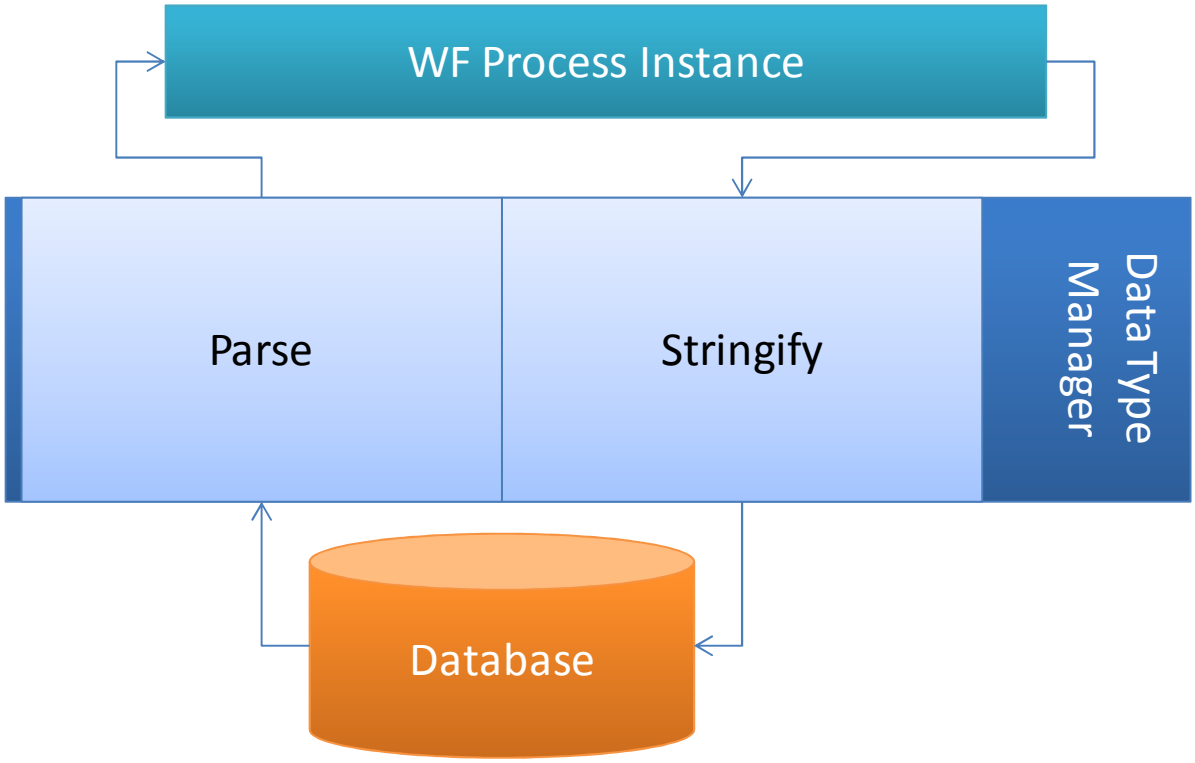

Figure A.4: The relationship between WF process instances and the data type manager.

Table A.7 lists all data types that are implemented within WEWE.

Table A.7: Data types supported by WEWE.

Data Type	*Description*
String	Normal JAVA boxed string object
Integer	Normal JAVA boxed integer object
Float	Normal JAVA boxed float object
Double	Normal JAVA boxed double object
Boolean	Normal JAVA boxed boolean object
Date	Using java.sql.Date library
Time	Using java.sql.Date library
Document/Reference	Using string to store reference to document. Eg: http://wewe.wits.ac.za/docs/application.pdf
Array	Using JAVA JSON Array Library
HashMap	Using JAVA JSON Array Library
Set	Using JAVA JSON Array Library
HashSet	Using JAVA JSON Array Library
Seconds	Custom class that provides time lapsed of WF process
Minutes	Custom class that provides time lapsed of WF process

Hours	Custom class that provides time lapsed of WF process
Days	Custom class that provides time lapsed of WF process

A.5 Conclusion

The blackboard is split into zones for each WF process instance which belong to a single workflow. Information in the WF process instances can be added, changed and removed by specialists. Changes made to the WF process instances are saved into the database.

A.6 References

[1] Concurrent HashMap (Java Platform SE 7). http://docs http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html, Last Accessed 6 January, 2014.

[2] Gawlick, Deiter, and Shailendra Mishra. Information sharing with the Oracle database. *In Proceedings of the 2nd international workshop on Distributed event-based systems*, pp. 1-6. ACM, 2003.

B. Specialist Implementation

B.1 Introduction

The implementation details of the specialist component of the workflow engine are provided. The manner in which a specialist is created from the front-end of the WFMS and persisted to the database is detailed. The way how specialists execute their actions is also detailed.

B.2 Specialist Creation and Persistence Implementation

The workflow process definition tool for WEWE is an internet application with a thin client that allows users to create specialists and their rules through a JavaScript imbued interactive interface. With very little assistance from the backend (server), the client interface uses JavaScript objects (more specifically JavaScript object literals) to temporarily store the information inserted by an administrator user regarding all specialists and their attributes. Once the user is satisfied with all the attributes of a particular specialist and inserts it into the definition of the workflow, all the specialist attributes stored in the JavaScript objects is serialized into a compact JSON string [1]. This JSON string consisting of the specialist rules accompanied with the specialist's metadata (name and description) is sent to the backend (server) of the process definition tool which then relays this information to the WEWE main engine to be stored as a record in Table B.1.

Table B.1: The schema for the table wewe_specialists.

Table "wewe_specialists"		
field	type(size)	default/options/indexes/constraints
specialist_id	serial	auto increment, primary key, not null
workflow_id	integer	foreign key (wewe_workflows:workflow_id)
specialist_name	varchar(100)	not null constraint
specialist_description	varchar(500)	not null constraint
specialist_rules	blob	not null constraint
specialist_status	integer	not null constraint
timestamp	timestamp	currentdatetime + timezone

From Table B.1, the specialist rules are saved as a JSON string in the 'specialist_rules' column. The format of this JSON string can be illustrated as an example in code listing B.1.

```
← - - Rule 1 - - →
IF (var.action_A == true AND const.name == "Salman Noor")
      <execute Action Set 1>
← - - End of Rule 1 - - →
← - - Rule 2 - - →
IF (var.action_B==true AND timelapsed.hours>=24) OR (var.action_C!=true)
            IF (const.studentNo=="0501698D")
                  <execute Action Set 2>
← - - End of Rule 2 - - →

CAN BE REPRESENTED USING JSON AS:

{ "attr1" : {
                  Rules : {    if1 : { ruleGroup1 : [
                                                { field : "var.action_A",
                                                  operator : eq,
                                                  value : true
                                                },
                                                {  field : "const.name",
                                                   operator : eq,
                                                   value : "Salman Noor"
                                                }
                                                   ]
                                        }
                           },
                  Actions : {<Execute Action Set 1>}
             },
"attr2" : {
                  Rules : {    if1 : { ruleGroup1 : [
                                                { field : "var.action_B",
                                                  operator : eq,
                                                  value : true
                                                },
```

```
                                                { field : “timelapsed.hours”,
                                                  operator : ge,
                                                  value : 24
                                                }
                                                    ]
                                        },
                                        { ruleGroup2 : [
                                                { field : “var.action_C”,
                                                  operator : ne,
                                                  value : true
                                                },
                                                    ]
                                        },
                                 },
                                {    if2 : { ruleGroup1 : [
                                                { field : “const.studentNo”,
                                                  operator : eq,
                                                  value : “0501698d”
                                                },
                                                ]
                                        },
                                 },
                        Actions : {<Execute Action Set 2>}
                }
}
```

Listing B.1: The format of the JSON string that stored specialist rules using an example.

From code block B.1, it can be seen that the format of the JSON string is ideal to store rules with nested if statements and antecedent conjunctions (AND) and disjunctions (OR). When a specialist is created, the specialist rules are loaded into the specialist object from this JSON string of rules stored in the database.

Actions to be executed in specialist rules are defined using JSON in the format shown in code listing B.2.

```
        {
                package_name : “The java package name”,
                class_name : “The name of class within the java package”,
                method_name : “Method to execute within the class”,
                argument_list : Array of arguments to be passed into the method,
```

```
	interaction_mode : Can be set to either user or auto
}
```

```
An example of an action to an application program that performs SQL queries
{
	package_name : “za.ac.wits.dbqueryexecutor”,
	class_name : “DBQueryPerformer”,
	method_name : “executeQuery”,
	argument_list : [ “mysql”, “localhost”, “3306”, “test_db”, “root”,
“pwd”, “insert into students values (‘Salman Noor’, ‘0501698d’)”
				],
	interaction_mode : auto
}
Will execute a function with the signature:


void executeQuery(String dataBaseType, String connIP, String connPort, String
DatabaseName , String username , String password, String SQLquery)
```

Listing B.2: The JSON format of a specialist action is shown.

From code listing B.2, each action defined should have an array of arguments that can be passed. An action can have no parameters and in this case an empty array of arguments will be passed. Code listing B.3 demonstrates how static and ad-hoc parameters can be defined.

```
An example of an action to an application program that performs SQL queries
{
	package_name : “za.ac.wits.dbqueryexecutor”,
	class_name : “DBQueryPerformer”,
	method_name : “executeQuery”,
	argument_list : [ “mysql”, “localhost”, “3306”, “test_db”, “root”,
“pwd”, “insert into students values (‘const.name’, ‘const.studentNumber’,
‘timelapsed.days’)”
				],
	interaction_mode : auto
}
Will execute a function with the signature:


void executeQuery(String dataBaseType, String connIP, String connPort, String
DatabaseName , String username , String password, String SQLquery)
```

Listing B.3: The JSON format of a specialist action is shown illustrating static and ad-hoc parameters.

From code listing B.3, the first six parameters (“mysql”, “localhost”, “3306”, “test_db”, “root” and “pwd”) are defined as static parameters. They cannot change whenever this action executed. The last parameter contains a SQL query which contains ad-hoc parameters (‘const.name’, ‘const.studentNumber’ and ‘timelapsed.days’) that will be retrieved at runtime from the blackboard. When the specialist instance is being animated by the specialist animator, it scans all actions for access specifiers (const, var, timelapsed, timeout) that indicate ad-hoc parameters and replaces them with values that can be found on the blackboard.

Actions in action sets have to be placed in specialist condition-action pairs (specialist rules) that are saved in JSON format. An example of this is given already in this Appendix. The JSON format of how actions are saved in a specialist rule is shown using examples in code listing B.4.

```
An example of an action set with a specialist rule:
{
       rules : {set of specialist rules},
       actions :    {
                            actionset1 : [
                                                {Action A}, {Action B}
                                                ],
                            actionset2 : [
                                                {Independent Action C}
                                                ],
                            actionset3 : [
                                                {Independent Action D}
                                                ]
                     }
}
```

Listing B.4: The JSON format of specialist actions within action sets.

From code listing 14, action set 1 contains two synchronous actions (Actions A and Actions B) where the latter action (Action B) might be dependent on the result of the first action. Action sets 2 and 3 each contain a single action that is set to be run independently.

Each application plug-in program needs to expose its actions (the methods) to WEWE. A common Java interface is defined that can be implemented by the applications plug-ins programs. This interface essentially provides a common contract between WEWE and the numerous application plug-in programs invoked by WEWE. Each application program must implement only one interface defined in code block B.1.

```
public interface AppProgram {

	public String getFeatureName();

	public void setSpecialist(Specialist theSpecialist);

	public void setWorkflowProcessData(WFProcess theWorkflowProcessData);

	public String getLastActionDoneDescription();

	public boolean lastActionSuccessful();

	public String getLastActionStatusCode();

	public String getFeatureShortDescription();

	public String getFeatureDetailedDescription();

	public void setInteractionMode(int theInteractionMode);
}
```

Listing B.5: The interface that needs to be implemented by invoked application programs.

Each application program can offer many actions (methods) with different return types, names and input parameters. It can be seen that the interface defined in code listing B.5 does not deal with the definition of actions. This is due to the difficulty in using an interface to implement multiple actions (methods) that can possibly have vastly different signatures (return types, names and input parameters). Furthermore, these actions are defined as normal methods. This presents WEWE with the difficulty of how to identify them as actions and invoke them. Java annotations were used to identify action methods and provide metadata (information about the action) which avert all difficulties mentioned. Appendix B3 will mention how Java annotations are utilized to execute specialist actions.

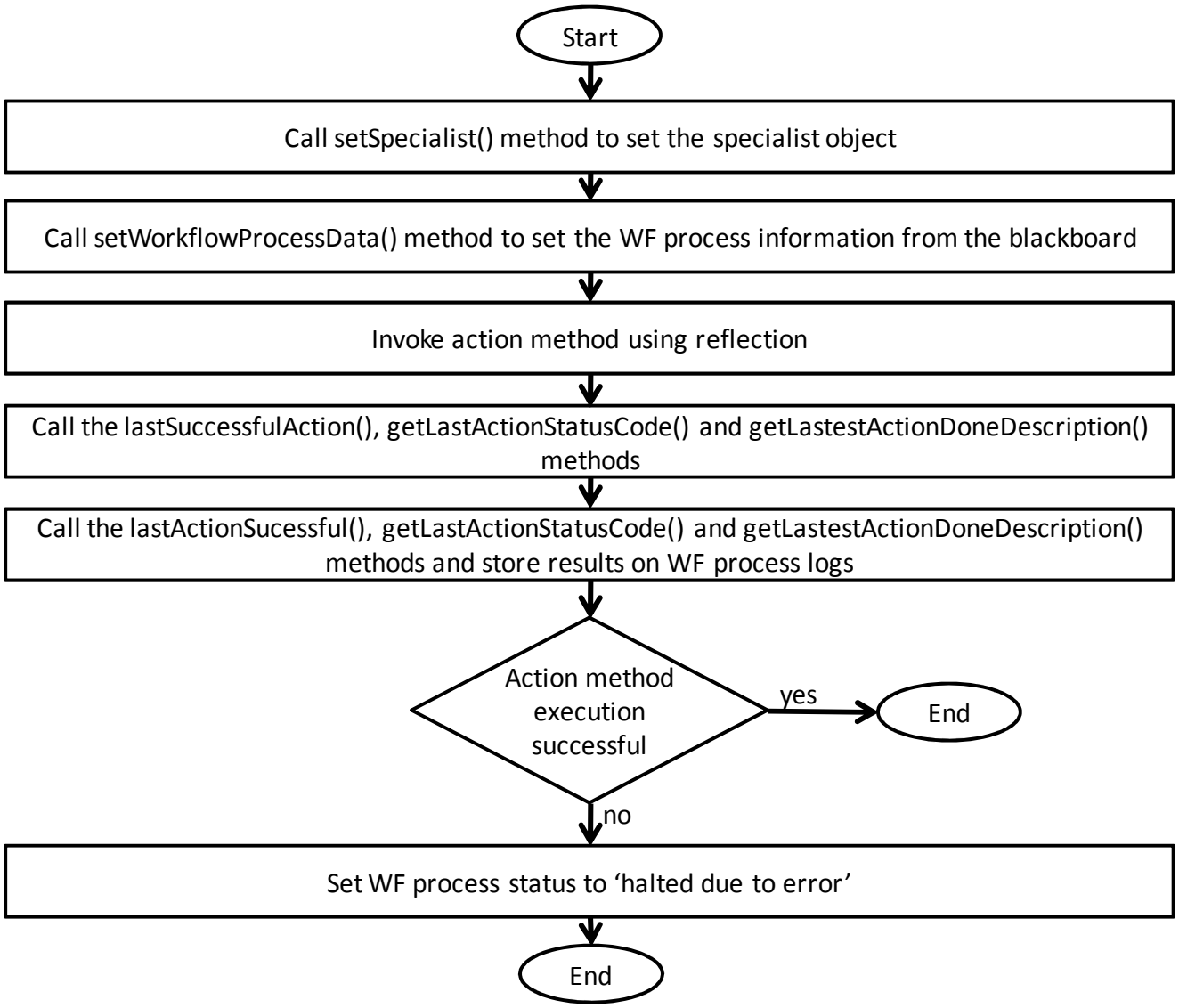

Figure B.1: The process undertaken by the specialist animator whenever an action is executed.

WEWE's specialist animator executes actions (from the application plug-in programs) from a set of actions defined as specialist rules (condition-action pairs). The specialist animator uses Java reflection to inspect the class that implements the AppProgram interface and invoke the relevant action method. The action might use and alter information from the WF process instance object from the blackboard. The action might also use specialist information such as the metadata of the specialist, what time the specialist became active, log information, etc. So before the specialist animator invokes that action method, it will invoke the `setWorkflowProcessData()` and the `setSpecialist()` methods so the action method can use the specialist and the blackboard (WF process instance object) information. Once the action method has been executed, the specialist animator will call the `lastestActionSuccessful()`, `getLastActionStatusCode()` and `getLastestActionDoneDescription()` methods in order and store the results of all these methods in the log component of the WF process instance object. Finally, if the action method did not execute successfully, the specialist animator will set the status of the WF

process (status is found on the blackboard inside the WF process instance object) to 7 (see Appendix A.3; status 7 means halted due to error). This process is depicted in Figure B.1.

B.3 Specialist Action Executor Implementation

Each application program can offer many actions (methods) with different return types, names and input parameters. It can be seen that the interface defined in code block B.2 does not deal with the definition of actions. This is due to the difficulty in using an interface to implement multiple actions (methods) that can have vastly different signatures (return types, names and input parameters). Furthermore, these actions are defined as normal methods. This presents WEWE with the difficulty of how to identify them as actions and invoke them. Java annotations can be used to identify action methods and provide metadata (information about the action) which avert all difficulties mentioned. The action annotation is defined in code block B.6.

```
The action interface is defined below:
public @interface Action {
	public String HumanReadableName();
	public String actionToBePerformed();
	public String[] argumentNamesActual();
}

The annotation defined above can be used to annotate action methods like so:

	@Action(actionToBePerformed = "Send email to one recipient only",
	argumentNamesActual = {"dearName", "content", "To", "From", "Message", "Bcc",
	"CC", "Subject","AttachmentURL"},
	HumanReadableName = "Send Email")
	public void sendEmail(String dearName, String content, String To, String From,
String Message, String Bcc, String CC, String Subject, String AttachmentURL){
//implementation of method goes here
}

	@Action(actionToBePerformed = "Send emails to multiple recipients including
	BCC, CC and attachements",
	argumentNamesActual = {"dearName", "content", "To", "From", "Message", "Bcc",
	"CC", "Subject","AttachmentURL"},
	HumanReadableName = "Send Multiple Emails")
	public void sendMultiEmail(String dearName, String content, String[] To, String
From, String Message, String[] Bcc,String[] CC, String Subject, String AttachmentURL)
{
```

```
//implementation goes here
}
```

Listing B.6: The interface that needs to be implemented by invoked application programs.

Using annotations, application programs can expose multiple actions by simply annotating the methods that execute these actions within a class that implements the AppProgram interface. For instance, an email application program can offer two actions which are sending a single email or sending multiple emails with BCCs and CCs (see code block B.6). Two methods with the action annotations can be used (see code block B.6) so WEWE can identify and invoke them.

B.4 Conclusion

The implementation details of how a specialist is created and persisted into the WFMS are detailed. Additionally, the implementation details of how a specialist animator executes actions offered by application programs are also detailed.

B.5 References

[1] How to: Serialize and Deserialize JSON Data. URL http://msdn.microsoft.com/en-us/library/bb412179. aspx, Last Accessed 18 March 2014.

C. WF Designer

C.1 Introduction

The workflow designer is a web-based WF client application that allows WF designers to create workflows. A flow of how a workflow can be created is provided with screenshots.

C.2 Creating a Workflow using WEWE's WF designer

The WF designer has a user management system that only allows WF designers to enter the system and create workflows. When a user tries to access the system on the web, the system will prompt users (WF designers) to enter their credentials before allowing them to enter the system as seen in Figure C.1.

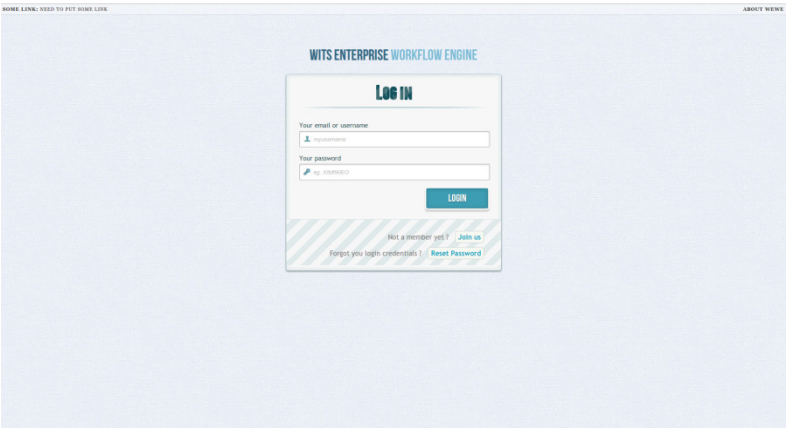

Figure C.1: Login Screen.

The home page of WEWE's WF designer allows users to create new and manage existing WFs as seen in Figure C.2. Screens are available (not shown in this appendix) to find analytical information about running workflow processes in the workflow engine. Users can manage these running WFs through the manage workflows interface. Additionally, users can alter running WFs if required by taking control of the orchestrator to alter WF data for these WF processes on the blackboard.

Figure C.2: Home page.

The appendix is however focused on creating new and editing existing WFs. To create a new WF, the user has to click the "workflows" menu item thereby opening a sub-menu. The user can click on the "design new workflow" item as shown in Figure C.3.

Figure C.3: How to start designing a new WF.

The first step in creating new WFs is providing information about the WF that is going to be created such as the name of the WF and some description (WF metadata) as shown in Figure C.4. In this example, a new WF named "Test Workflow" is being created. Once this information is set, users can always change this information by selecting "edit completed workflow" option within the "workflows" menu shown in Figure C.3. To set the WF metadata, the "set metadata" button needs to be clicked thereby progressing to the next step in creating WFs.

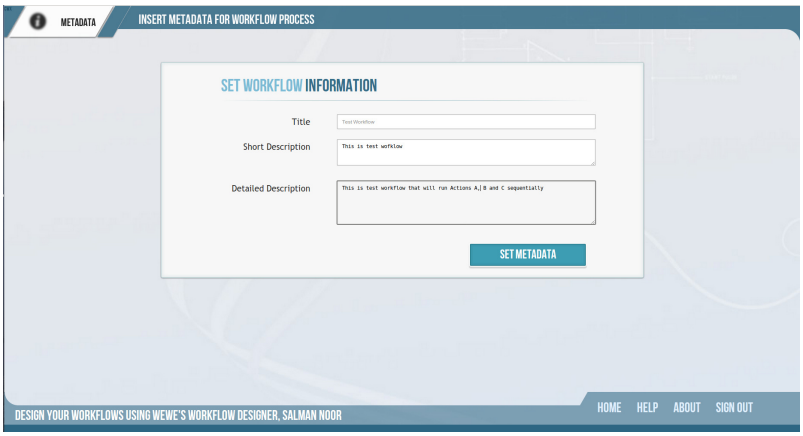

Figure C.4: Screen for inserting WF metadata.

The next step in creating WFs is defining the constraints and dependencies of the WF in order to make certain that enough initial correct information is available to orchestrate every WF process instance of this WF successfully to completion. Initial WF constants and variables (shown as 'parameters' in the WF designer) can be defined. Additionally, workflow objects can be created and defined that allows users to create custom constraints on the WF data. The screen that allows users to define these constraints and dependencies is shown in Figure C.5.

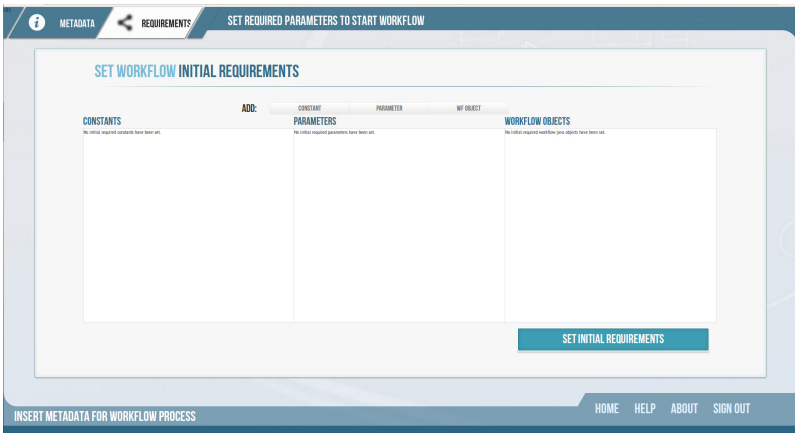

Figure C.5: Screen for setting WF dependencies and constraints.

As an example for this "test workflow", one initial constant is defined by clicking on the "Add Constants" button shown in Figure C.5 which opens up a modal window to define and create a new initial constant dependency shown in Figure C.6. A new initial constant called "surname" is defined. This essentially means that when a new instance of this WF is created, a constant named surname needs to already exist in the WF data for the WF process to run successfully.

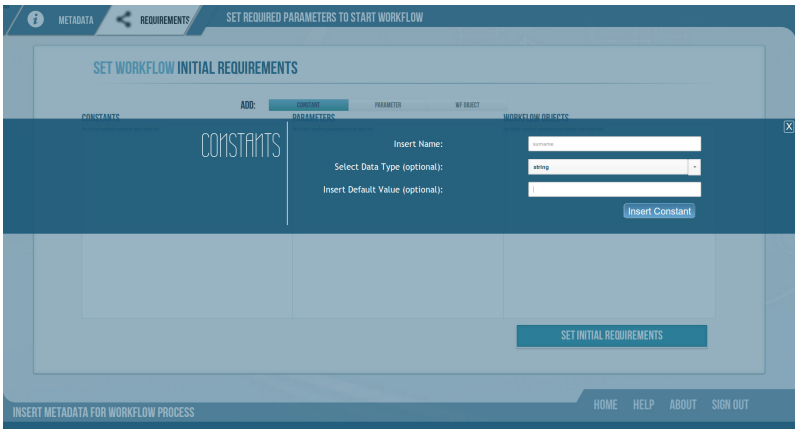

Figure C.6: As an example a screen to insert a new constant WF constraint.

From Figure C.6, the new initial constant can be defined by clicking on the "insert constant" button, which will close the modal window and add this newly defined initial constant to the constants list as shown in the Figure C.7. The administrator can also edit and remove already defined dependencies and constraints by simply selecting any item on the lists which opens up a modal window that provides options to edit or delete them.

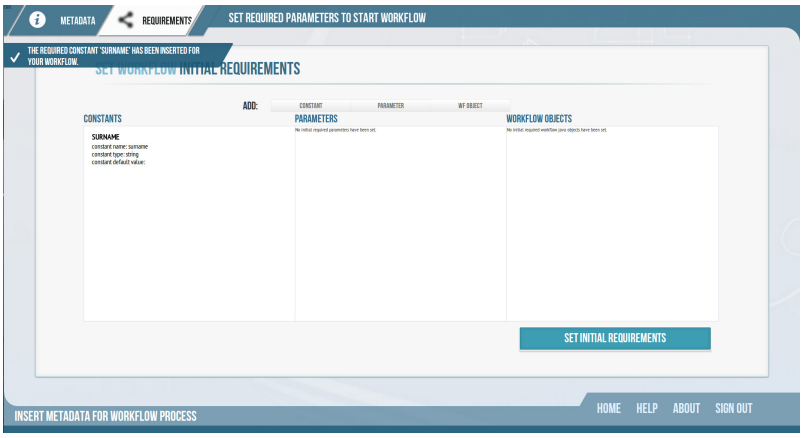

Figure C.7: As an example a screen to new constant being defined as a WF constraint.

The next stage of the WF creation process can now be reached of defining web interfaces for the WF when the "set initial requirements" button is clicked in Figure C.7. Figure C.8 shows the screen that allows users to define HTML web forms as interfaces at different stages of the WF. Additionally, for each interface, agents are defined that transport data submitted from the defined form. However, defining interfaces is not necessary for all workflows as some workflows can be stimulated by other interfaces (not necessarily HTML web forms) and external systems that can post data to the blackboard. By clicking on the "set interfaces" button, the user can proceed to next stage in creating a WF.

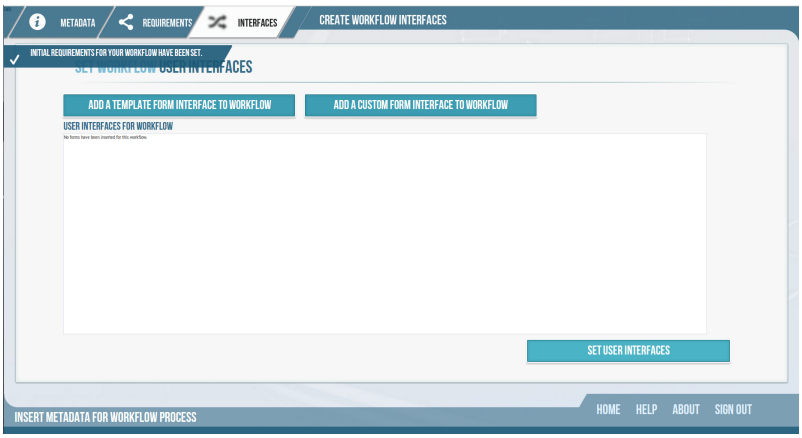

Figure C.8: The screen to insert interfaces for the WF.

The next and most important step is to define specialists' definitions for the WF as shown in Figure C.9. Typically, a single specialist is created for every single WF task or activity. To create a brand new specialist, users can click on the "create new specialist" button.

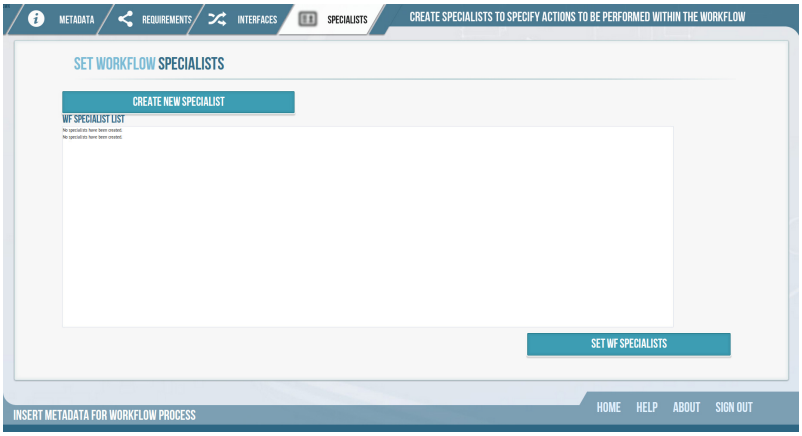

Figure C.9: The screen to create specialists for the WF.

A modal window will open that allows the user to insert metadata (the name and description) for the new specialist as shown in Figure C.10. As an example, a specialist named "Specialist A" that will execute a hypothetical action A will be created.

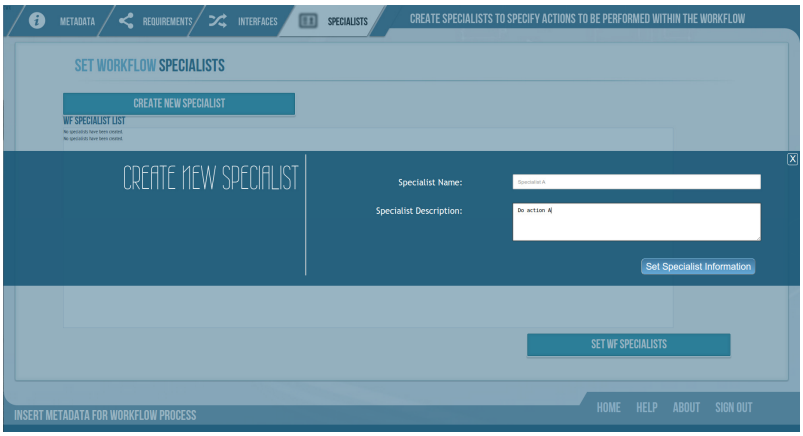

Figure C.10: The screen to insert metadata for the new specialist.

Once the user clicks on the "Set Specialist Information" button as shown in Figure C.10, a new screen will open up with a specialist main menu which is shown in Figure C.11. This main menu will allow the user to set any information regarding this single specialist. The first menu item, "Specialist Information" simply allows the user to edit the information inserted in Figure C.10. The second menu item, "specialist status" allows the user to simply disable or enable the specialist within the workflow. This option is given to the administrator to provide more flexibility when creating workflows. Several scenarios can arise. Perhaps, the administrator might not want to enable the specialist that has been created since the specialist has not been completed fully or the requirements of the workflow is not fully completed to warrant the inclusion of this specialist to the WF. The last menu item "Specialist Timeout" allows the user to set the timeout information if needed and define what actions to perform when the timeout occurs.

The most important menu item is "specialist attributes" which allows the administrator the define condition-action pairs (specialist rules) for the specialist.

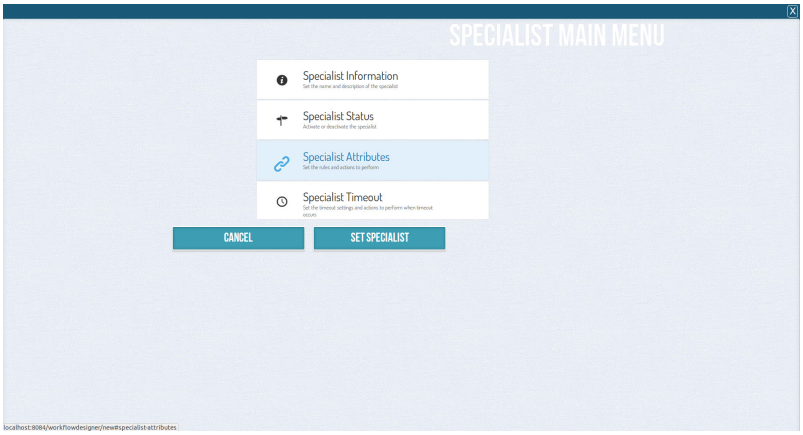

Figure C.11: The specialist main menu.

By clicking on "specialist attributes" in Figure 54, the administrator enters the specialist attributes menu shown in Figure C.12. Since, there are no attributes already defined; only one option that is the "add specialist attribute" exists. Once, a specialist attribute is defined, other options will be available such as to inspect, edit and delete the specialist attribute. Additionally, many specialist rules can be added from this menu. Once the user has created all of the specialist attributes for this specialist, the user will click on the "set specialist attributes" button. For this example, the administrator will click the "add specialist attribute" menu item to create an attribute.

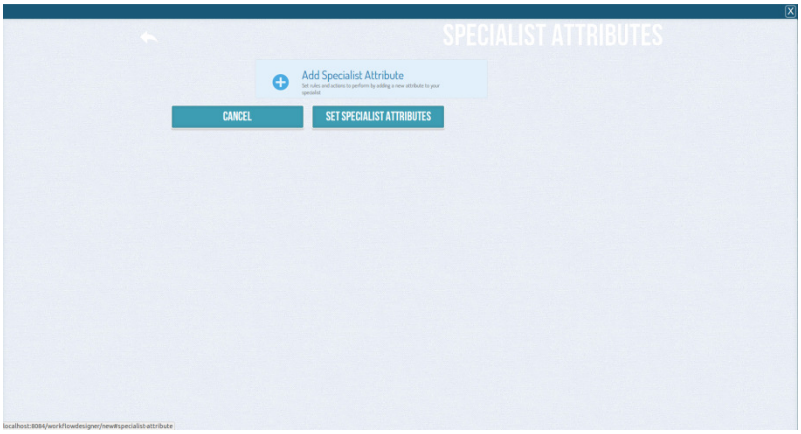

Figure C.12: The specialist attribute main menu.

To create a specialist attribute, the user has to set the rules and actions to execute if those rules are evaluated to be true as seen in Figure C.13.

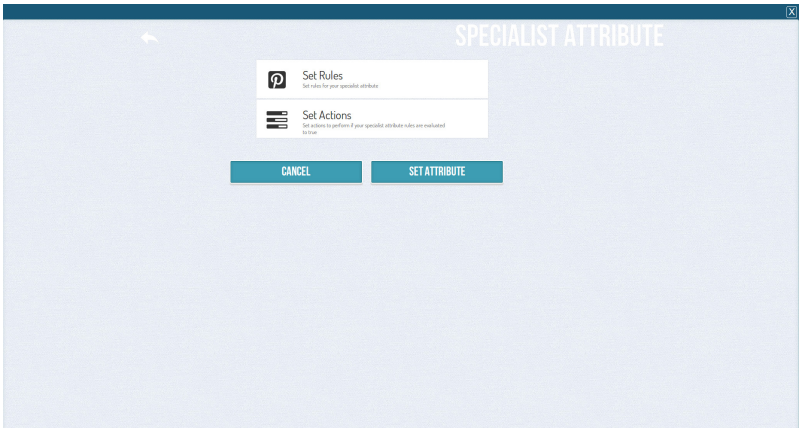

Figure C.13: The specialist attribute creation menu.

The screen in Figure C.14 allows users to insert the rules for the specialist attribute. On the left hand side, it can be seen that there is a list of rule antecedents and on top of the list is a button named "add rule" which can be clicked to add new antecedents to the list.

Figure C.14: The specialist rule creation screen.

If the user clicks on the "add rule" button in Figure C.14, a modal window will open up to allow the user to define and create a new antecedent as shown in Figure C.15. As an example, a rule shown in Figure C.15, an antecedent is created to check if the variable (WF designer shows variables as parameters) "action_a_done" is true. Any antecedents can be created from this modal window. When the user clicks on the "set condition statement" button, this antecedent is added to the antecedent list as shown in Figure C.14 on the left hand side of the screen.

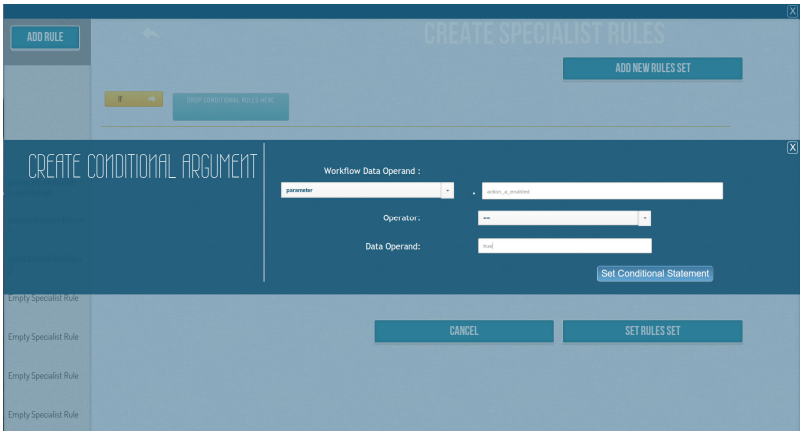

Figure C.15: The specialist rule antecedent modal window screen.

Once all antecedents are added to the list, the user can simply drag and drop them into the rule (next to the yellow "if" statement) as shown in Figure C.16. Many antecedents can be added in any order from the antecedent list.

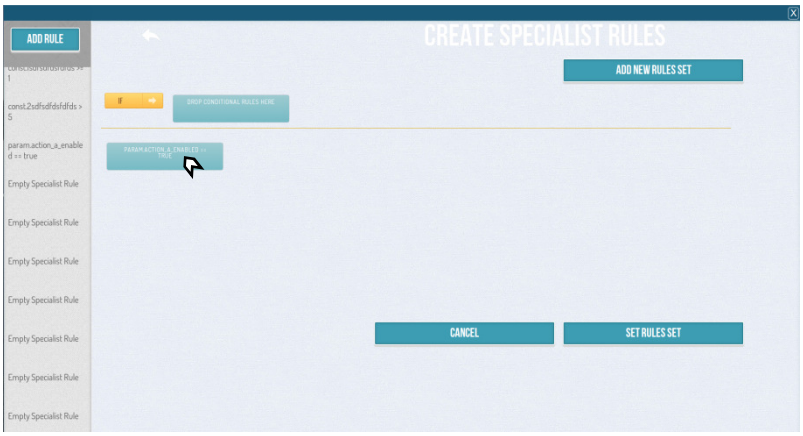

Figure C.16: Antecedent drag and drop screen.

The condition can be made up of many antecedents that can be joined by the 'AND' and 'OR' keywords. The user can set these 'AND' and 'OR' keywords between any two antecedents using the blue button as seen in Figure C.17. In Section 5.3.2, the 'AND' keyword (conjunction) is used to conjoin antecedents together and the 'OR' keyword (disjunction) is used to separate antecedents. If the user wants to reverse 'AND' and 'OR' conventions, the user can click on "Add new Rule Set" to add nested 'IF' statements to fabricate an antecedent evaluation result identical to the result if the keyword conventions are reversed.

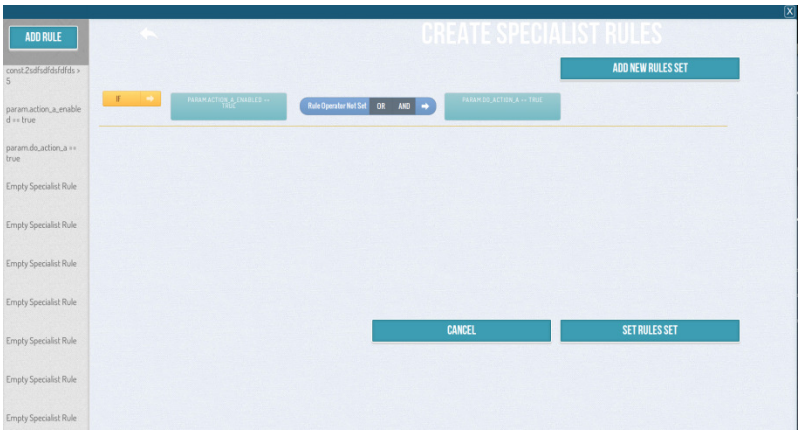

Figure C.17: Setting antecedent conjunctions.

In this example, the "AND" keyword is selected to conjoin two antecedents within one 'if' statement as seen in Figure C.18. Once the user has set all the rules for this attribute, the "set rules set" can be clicked after which the system will return to the menu shown in Figure C.13.

Figure C.18: An example of selecting the 'AND' keyword.

From Figure C.13, the user can now select the "set action" menu item to enter the screen shown in Figure C.19. Similar to the rules creation screen, a list also exists on the left hand side of all the actions. The user can click on the "add new action" button to add new actions to the list.

Figure C.19: The actions creation screen.

The user clicking the "add new action" button will open a modal window shown in Figure C.20. From the modal window, a content slider contains all the application plug-in programs that are inside the WFMS. The user can scroll the content slider to select the required application plug-in program. Once an application program is selected, a list of actions of that application plug-in program is provided for the administrator to select. As an example, the user has selected the "Debugger" application program that basically logs actions to a text file with a single action called "log message". The user can click on the "log message" action and enter parameters for that action as seen in Figure C.21.

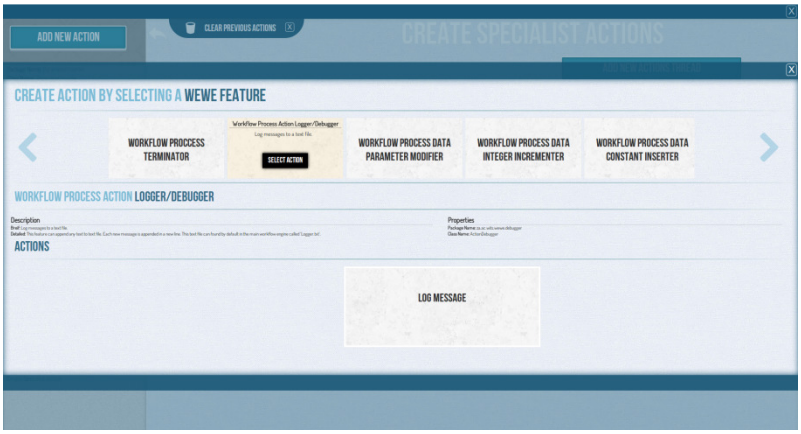

Figure C.20: The action creator modal window.

From Figure C.21, it can be seen that the "logger action" has only one parameter that can be set which is the actual log message that needs to be written to a file. The user can insert the relevant text and set this action which will be inserted into the actions list on the left side of the action creation screen shown in Figure C.19.

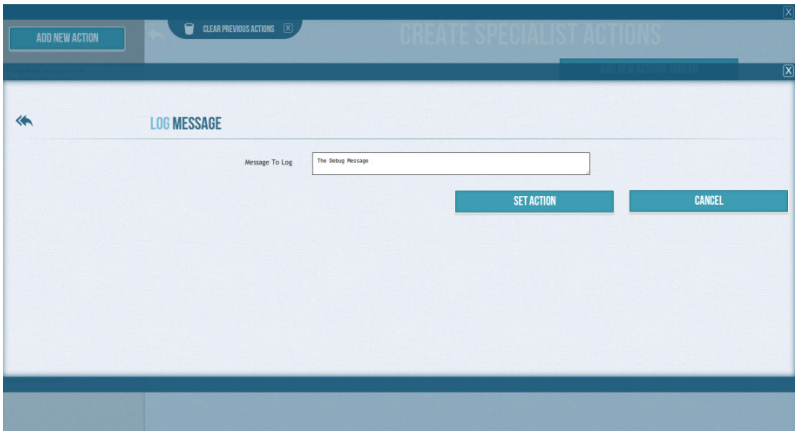

Figure C.21: The action parameters setter modal window.

Similar to the rules creation screen, actions can also a dragged from the action list and dropped into an action thread. In Section 5.3.5, specialists were designed to run actions asynchronously or synchronously. A set of synchronous actions run consecutively and may require the results from preceding actions in order to run. Asynchronous actions run in their own thread and can be made to run independently from the executions of other actions. From Figure C.22, a user can add many action threads by clicking on the "add actions thread" button. Within these action threads, the user can drag and drop actions in the desired order. As an example in Figure C.22, two action threads have been created. The first action thread has two actions that are executed in order and the second action thread contains a single action.

Figure C.22: The drag and drop action function.

When the user has created the rules and actions for the specialist attribute, the specialist attribute menu shown in Figure C.23 changes to accommodate the editing and deletion of rules and actions. If the user is satisfied with the specialist attribute, the user can click on the "set attribute" button. The user can always update this specialist attribute at a later time. Furthermore, many specialist attributes can be created, updated and removed in a like manner for a single specialist.

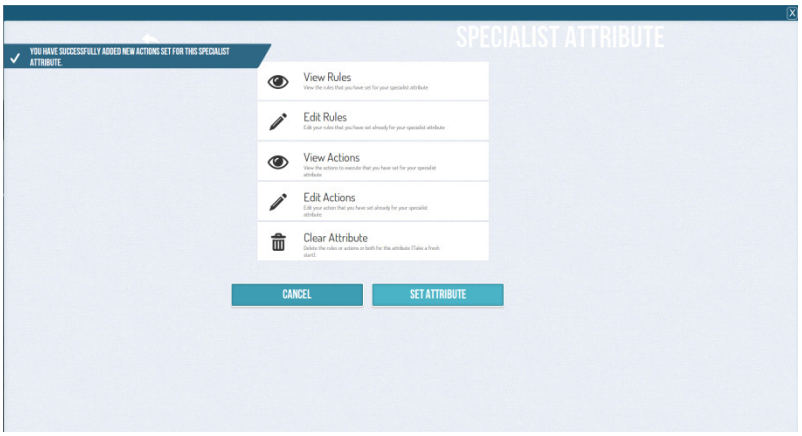

Figure C.23: The specialist attributes menu when attributes have been added.

After setting all the specialist attributes, metadata and timeout information, the user can click on the "set specialist" button shown in Figure C.13. This will close the specialist menu and add a new specialist to the workflow as shown in Figure C.24. As an example, specialist A is created and can be seen in the list. To edit the created specialist, the user simply double clicks on the specialist item in order to update or remove the selected specialist.

Figure C.24: The specialist screen with one specialist created.

Once specialists have been created for the workflow, the process of creating the new workflow is completed and the user will return to the home screen shown in Figure C.1.

C.3 Conclusion

The entire flow of how a workflow is created has been shown with screenshots from the actual WF designer. The process entails the insertion of metadata, initial dependencies and constraints, interfaces and specialists.

D. WF Example to Demonstrate WF Orchestration

D.1 Introduction

As discussed briefly in Section 5.7 of the main body of the dissertation, an example will be used of an insurance claim processing workflow (shown in Figure D.1) to demonstrate how WEWE built on the blackboard paradigm will orchestrate this workflow. This will provide the reader of a better holistic understanding of the workflow engine. This Appendix is dedicated to provide detailed information about how this workflow process is actually orchestrated in WEWE.

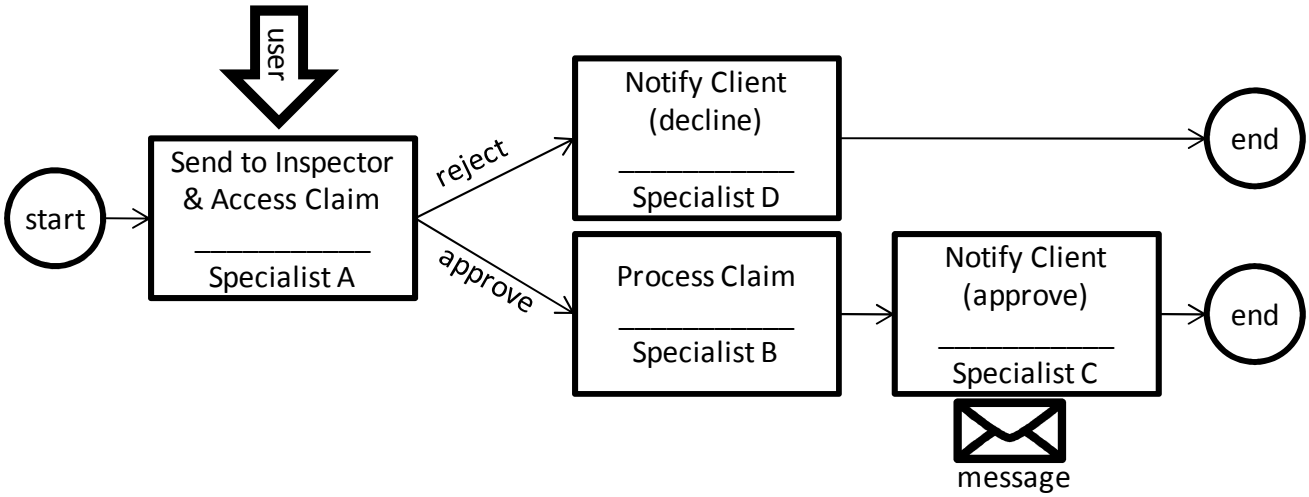

Figure D.1: Hypothetical workflow of approving or rejecting an insurance claim.

D.2 Designing the Specialists

It is found through experimentation that specialists should be created for every single atomic workflow activity. It is possible that all the workflow activities within the workflow can be put inside one single specialist as many rules. However, this prevents the concurrent execution of workflow activities that are required to be executed concurrently. Furthermore, blackboard data write conflicts will occur. Therefore, many modular specialists should be created for concurrency and to prevent data conflicts on the blackboard.

D.3 WF Proccess Orchestration

This section will deliver detailed information about the WF process orchestration which will include the state of the blackboard and the interactions of the specialist and agents that change the state of the blackboard at every stage of the workflow.

D.3.1. WF Spawn

When a new WF process is spawned, a new WF process object is created by the orchestrator and placed on the blackboard with the following data as seen in Figure D.2. It can be seen that the production data is stored as constants so specialists can simply use data and not alter it in any way. All the constants are stored as Plain Old Java Objects (POJOs) containing information about the insurance claim; the client's information and the insurance inspector's information that the specialists can use to execute their actions.

Blackboard	
Variables	
enable_assess_claim	True
Constants	
Insurance Claim Information	WF Object (POJO)
Client Information	WF Object (POJO)
Insurance Inspector Information	WF Object (POJO)

Figure D.2: Initial blackboard data when the process is spawned.

From the blackboard in Figure D.2, there is a flag to enable the assess claim action but no trigger flag to actually execute the action, since it is user triggered. The first rule of Specialist A has an action to send the claim information to the claim inspector when the 'assess claim' action is enabled. This is seen in Figure D.3 as only the first of the three rules is actually executed. The action of this rule does not change the content of the blackboard but uses the content on the blackboard specifically all the production data (constants) in order to send the insurance claim information to the inspector.

D.3.2. Access Claim Activity

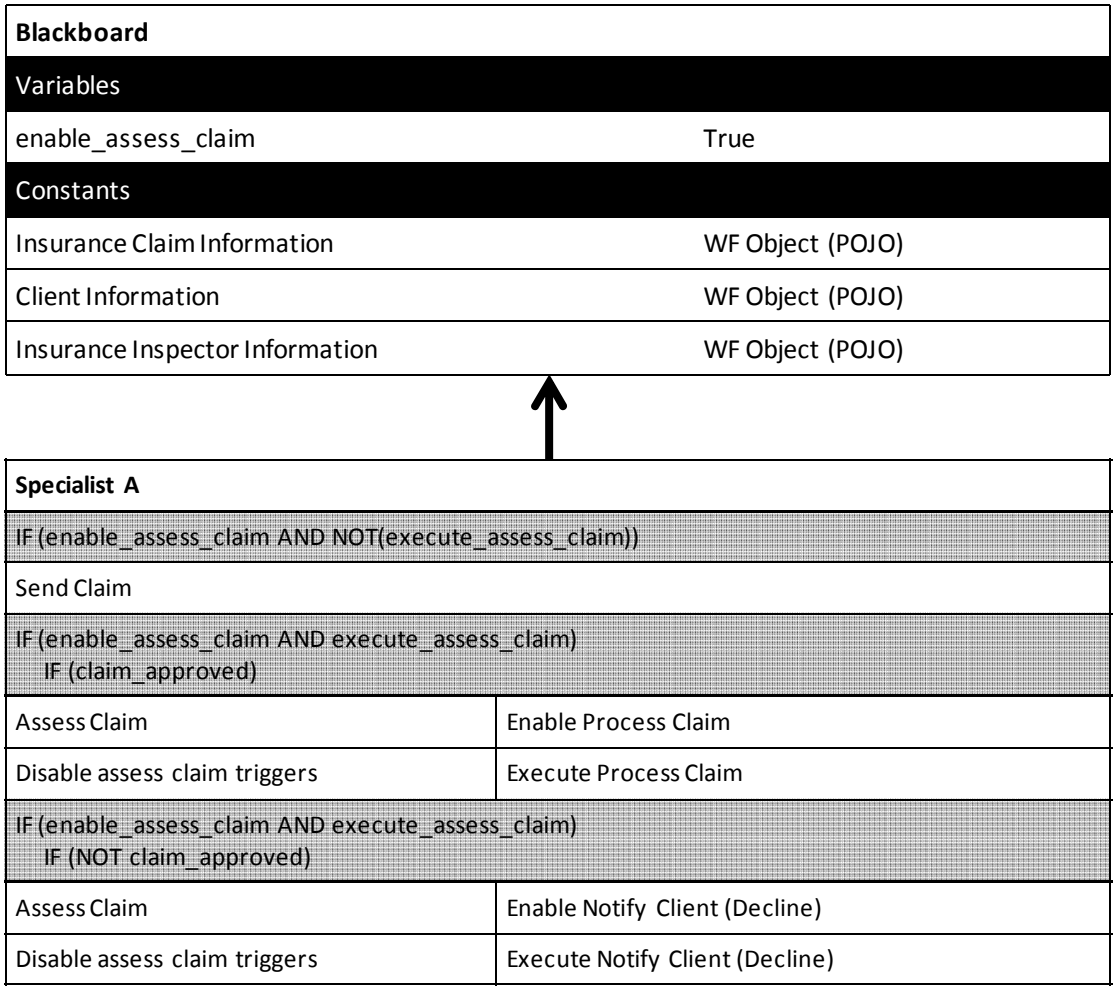

Figure D.3: Specialist A interacting with the blackboard.

The inspector will assess the claim through a client application within the WFMS. When the inspector accepts or rejects the claim, agents are created by the client application that will post data to the WF process object on the blackboard as seen in Figure D.4. In this example, the inspector approves the claim. Two agents will post the data on the blackboard; one will post the approval and the other will post a trigger flag for executing the 'assess claim' action. This change on the blackboard will enable the specialists to start monitoring the blackboard.

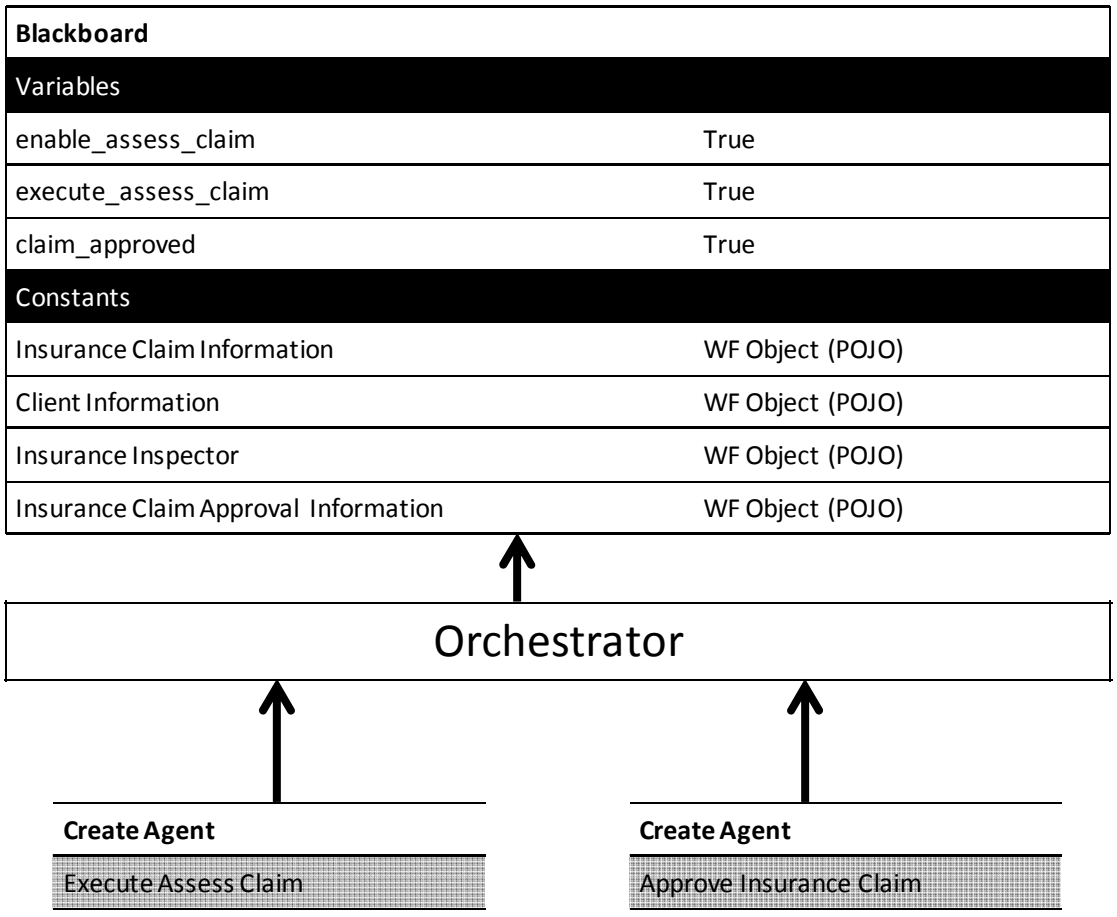

Figure D.4: Agents interacting with the blackboard to trigger an action.

Specialist A monitoring this WF process object on the blackboard will notice that 'execute_assess_claim' trigger flag exists. Specialist A will execute its second rule that will be executed when the inspector approves the claim. The third rule will never be executed in this example since it is only executed when the inspector rejects the claim. The second rule consists of the 'assess claim' action which is an action to log the inspector's actions on a tracking system. It can be seen that there are two separate action threads in this second rule. While the specialist is executing the current action of logging the access claim details on the tracking system, it simultaneously enables and triggers the next action to 'process claim'. Notice that the next action of 'process claim' is automatically enabled and triggered so it can automatically be executed after the 'assess claim' action. After the 'assess claim' action is executed, the specialist also disables the triggers to the 'enable_assess_claim' and 'execute_assess_claim' actions (changed to the Boolean, false) which is crucial. If this is not done, Specialist A will keep on executing its second rule. Specialists respond to triggers on the blackboard. If the triggers remain unchanged, this workflow will be locked in a certain state that might only execute one specialist (one WF activity) repeatedly.

Blackboard	
Variables	
enable_send_claim	False
execute_sent_claim	False
enable_assess_claim	False
execute_assess_claim	False
claim_approved	True
enable_process_claim	True
execute_process_claim	True
Constants	
Insurance Claim Information	WF Object (POJO)
Client Information	WF Object (POJO)
Insurance Inspector	WF Object (POJO)
Insurance Claim Approval Information	WF Object (POJO)

Specialist A	
IF (enable_assess_claim AND (NOT execute_assess_claim))	
Send Claim To Inspector	
IF (enable_assess_claim AND execute_assess_claim) IF (claim_approved)	
Assess Claim	Enable Process Claim
Disable assess claim triggers	Execute Process Claim
IF (enable_assess_claim AND execute_assess_claim) IF (NOT claim_approved)	
Assess Claim	Enable Notify Client (Decline)
Disable assess claim triggers	Execute Notify Client (Decline)

Figure D.5: Specialist A interacting with the blackboard for the second time.

D.3.3. Process Claim Activity

Specialist A has placed the triggers on the blackboard to execute the WF activity in Specialist B which is to process the claim as seen in Figure D.5. Specialist B will execute its first rule and process the claim. Thereafter, the triggers to process the claim will be disabled so that Specialist B will not execute its rules repeatedly. While Specialist B is executing its actions, it will enable the next WF activity which is to notify the client that the claim has been approved. This is shown in Figure D.6.

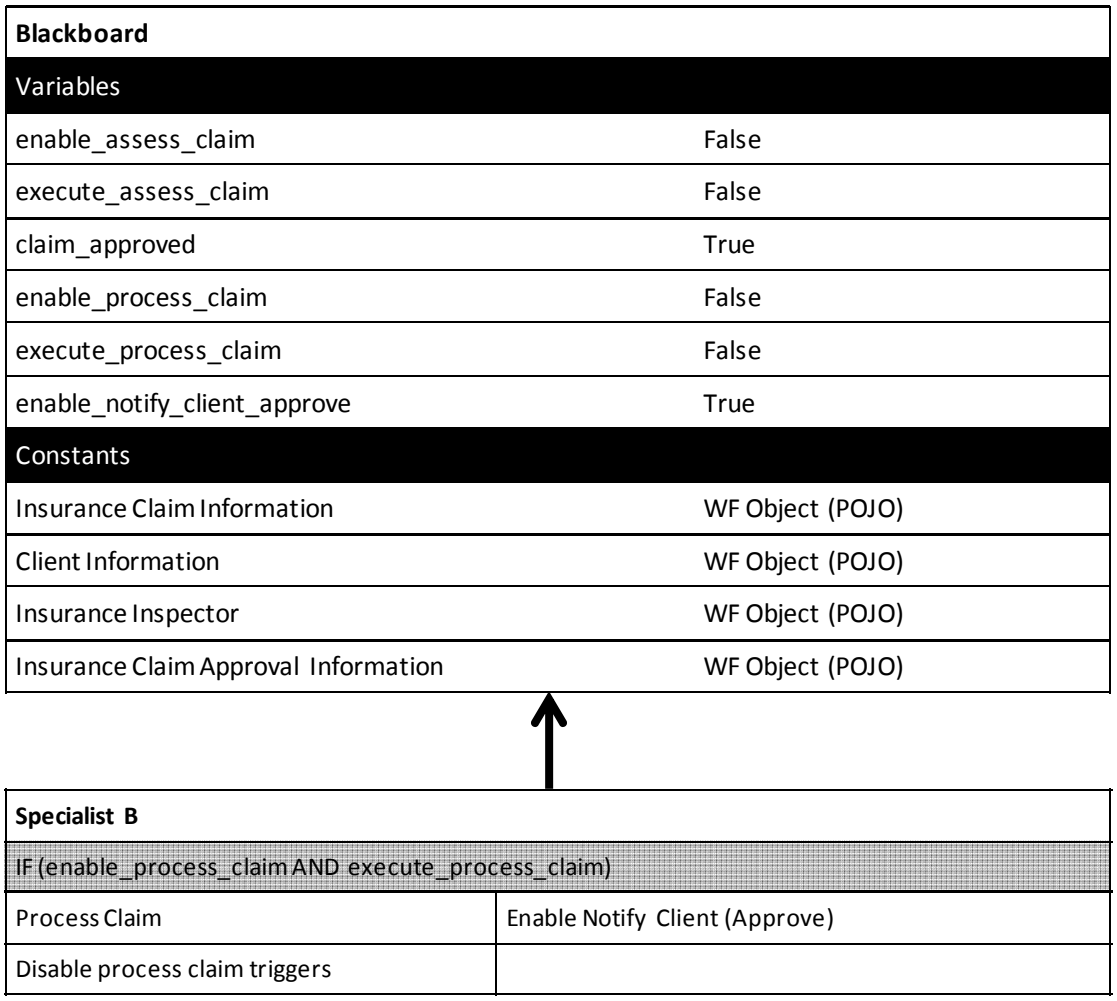

Figure D.6: Specialist B interacting with the blackboard.

Specialist B does not place the actual trigger to execute the next WF activity of sending an approval message to the specialist. The WF activity will only be triggered by a message delivered by an external system indicating the claim has been successfully processed. The software infrastructure that is implemented might not be able to process the claim immediately. For instance, a batch process might execute at the end of every business day that will process all approved claims at once. Another scenario would be placing the approved claim into a queue that processes claims in FIFO order. Once the external system has processed the approved claim, it will send an agent to put a trigger flag to enable another specialist to engage with the blackboard. This is shown in Figure D.7.

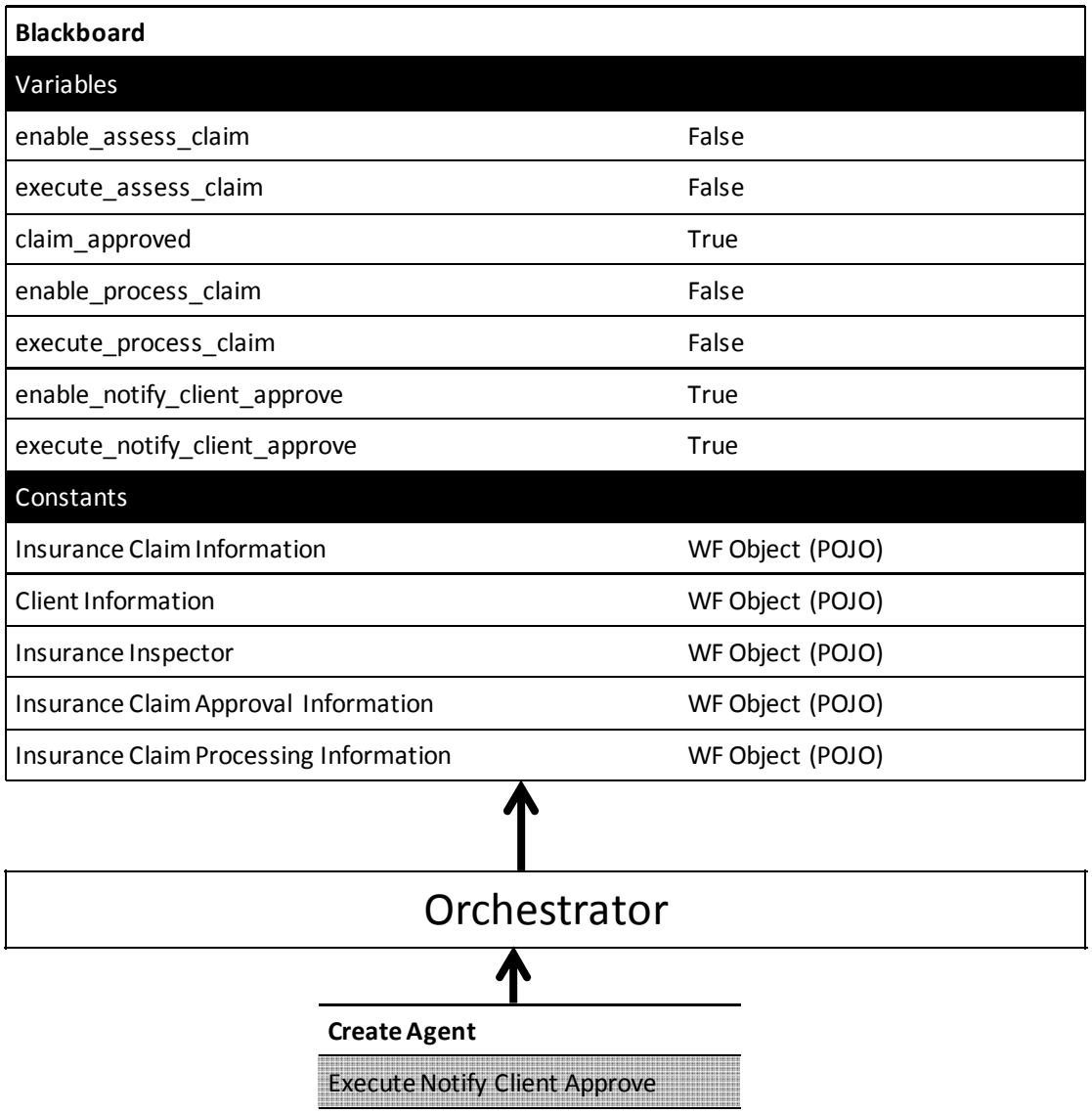

Figure D.7: An agent interacting with the blackboard to trigger the notify client action.

D.3.4. Notify Client Activity

Once the external system successfully processes the claim, it will send an agent with a trigger flag to place on the blackboard (execute_notify_claim_approve) so that Specialist C can execute its first rule to execute the activity to notify the client that the claim has been approved. This is shown in Figure D.8. The specialist will execute the action to notify the client via email, fax or SMS depending on the choice of the client when the claim was submitted by the client to the WF engine. After this action has been executed, the trigger flag to execute the notify client WF activity will be disabled. The last action of this Specialist is to terminate the workflow process. The application program that the specialist invokes to terminate the workflow will prompt the orchestrator to terminate the workflow as discussed in Section 5.5.1.

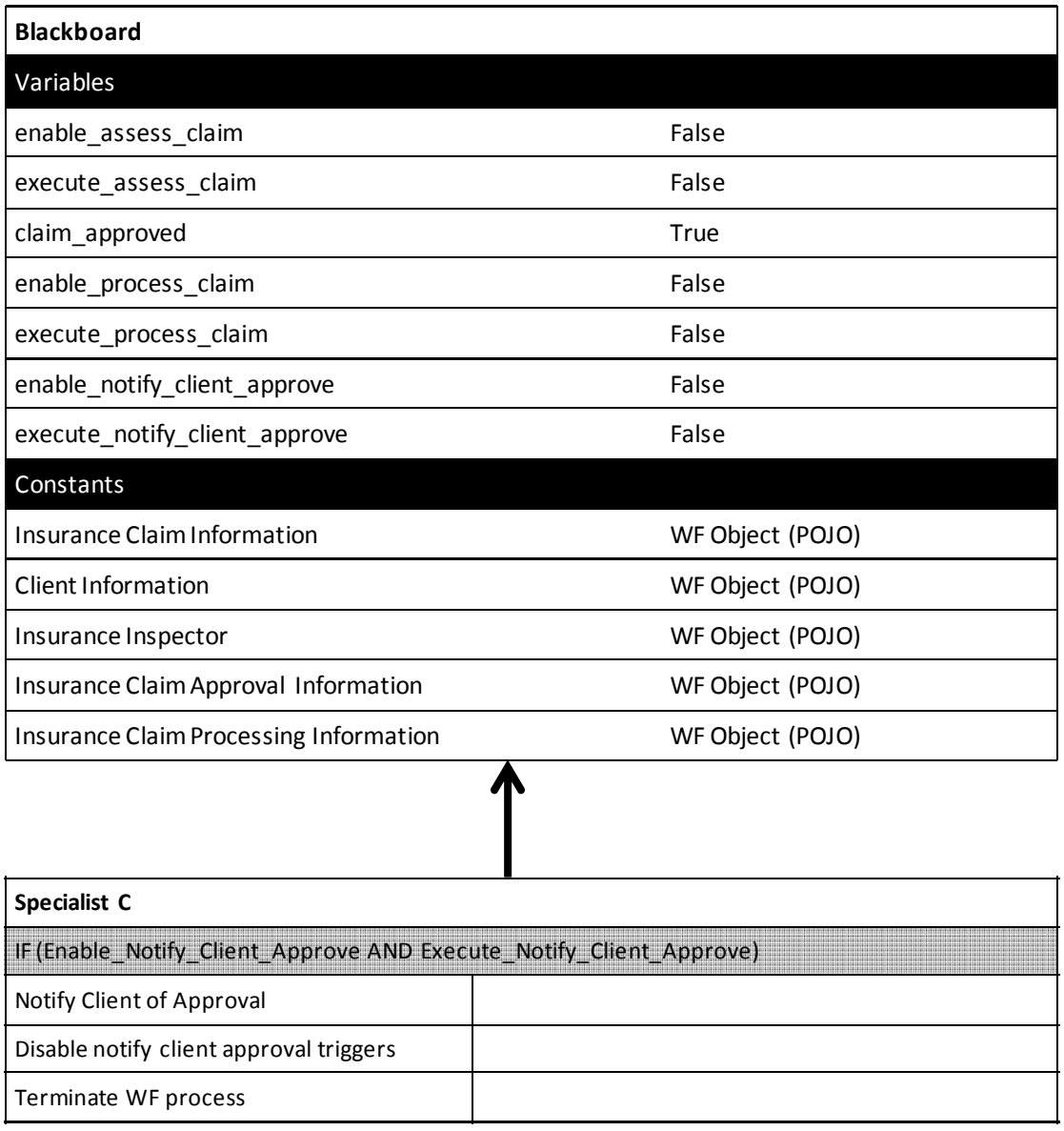

Figure D.8: Specialist C interacting with the blackboard to execute the last WF activity in the WF.

Throughout the workflow process, the WF production data in the shape of constants on the blackboard were used to execute specialist actions. The specialist action to notify the client that the claim has been approved requires the contact details (email address, contact number, etc) to be able to execute this action. The process claim action will require most of the production data. Typically production data is placed as constants on the blackboard to prevent any changes to the data. However depending on the application, some workflows might require production data to be altered in the course of the WF process. The control data (triggers to WF activities) should be saved as variables that can be altered at any time. Production data can be variables too depending on the requirements of the workflow.

D.4 Conclusion

A meticulous WF action to action explanation of how blackboard system is used to run a workflow has been provided. Figure 33 in main dissertation shows a high level representation of this in a form of sequence diagram.