Automated Unit Testing of Agent Systems

A thesis submitted for the degree of
Doctor of Philosophy

Zhiyong Zhang,
School of Computer Science and Information Technology,
College of Science, Engineering and Health,
RMIT University,
Melbourne, Victoria, Australia.

October 14, 2011
Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged; ethics procedures and guidelines have been followed.

Zhiyong Zhang
School of Computer Science and Information Technology
RMIT University
Acknowledgments

I thank my supervisors Prof. Lin Padgham and Dr. John Thangarajah. My research and this thesis would not have been finished without their help. It is difficult to express my gratitude to them. As supervisors they both have guided my development as a researcher over the last few years. The knowledge and skills I have gained with their supervision are priceless. In particular, I thank Lin for helping me obtain scholarships, which have been of great value to me.

A special thank you to my wife Huanglu Wu, without whose encouragement and supports I could not persist in my research until the submission. She has suffered great amounts of sacrifice separating with me for over 2 years and we had to communicate via international phone calls. Although she is not with me in Australia, she has helped me significantly in my thesis work and my life with extremely valuable advice and ideas. She has assisted me in my thesis writing by providing suggestions on various aspects. I would be grateful to my wife for the rest of my life, for ever.

I would also like to thank my parents for their sacrifices to support my education, their loving care when I was with them for brief visits.

During my thesis writing Li Mei Brusey has assisted me significantly by providing educational feedback and suggestions. I thank her for all she has done to help me improve my writing skills and my thesis quality.

I also acknowledge the co-authorship and contributions of A/Prof. Michael Winikoff and Dr. David Poutakidis in my publications.

I thank all members of the Agent Group in RMIT. Academic activity in the group has made me learn extra knowledge and principles beyond my research discipline.

Thanks to you all!
Credits

Portions of the material in this thesis have previously appeared in the following publications:


Contents

Abstract 1

1 Introduction 2

2 Background 8
   2.1 Agents ......................................................... 8
   2.2 AOSE Methodologies ........................................... 12
   2.3 Prometheus .................................................. 14
   2.4 Software Testing ............................................. 22
       2.4.1 An Introduction to Testing ............................... 23
       2.4.2 Unit Testing ........................................... 26
   2.5 State of The Art ............................................. 30
       2.5.1 Unit Testing Strategies ................................ 30
       2.5.2 Test Automation ........................................ 33
       2.5.3 Model Based Testing ................................... 37
       2.5.4 Testing Agent Systems ................................. 41

3 Test Units 46
   3.1 Units in Agent Systems ..................................... 46
   3.2 Fault Model ................................................ 48
       3.2.1 Plans .................................................. 50
       3.2.2 Plan Cycles ........................................... 53
       3.2.3 Events ................................................ 55
       3.2.4 Beliefs ................................................ 57
       3.2.5 Concurrency Discussion ............................... 64
## CONTENTS

4 Testing Framework
   4.1 Testing Process Overview ........................................... 67
   4.2 Order of Testing .................................................... 70
      4.2.1 Dependencies between Units ................................. 71
      4.2.2 Algorithm ...................................................... 74
   4.3 Testing Harness ..................................................... 80
      4.3.1 Structure of Test Harnesses ................................. 80
      4.3.2 Unit Testing Process ........................................ 85
      4.3.3 Initialisation Procedures ................................. 87
   4.4 Code Augmentation .................................................. 91
      4.4.1 Plan ......................................................... 96
      4.4.2 Plan Cycle ................................................... 103
      4.4.3 Event ....................................................... 104
      4.4.4 Belief ....................................................... 106
      4.4.5 Implementation of Mock Agents ............................ 108
   4.5 Test Report ......................................................... 112

5 Test Input Generation .................................................... 118
   5.1 Input Variable Declaration ....................................... 120
   5.2 Test Input Generation ............................................. 124
      5.2.1 Manual Test Case Input .................................... 128
   5.3 Comparison between Variables ................................... 129
   5.4 Variable Assignment ............................................... 134
   5.5 Specification of False Positives ............................... 138

6 Evaluation ............................................................... 140
   6.1 Evaluation Objective .............................................. 140
   6.2 Experimental Process ............................................ 141
   6.3 Application Systems for Evaluation ............................ 143
   6.4 Fault Analysis and Evaluation .................................. 144
      6.4.1 Fault Distribution ......................................... 146
      6.4.2 Fault Categories and Fault Types ....................... 147
      6.4.3 False Positives .............................................. 149
      6.4.4 Testing Process Evaluation ............................... 153
## CONTENTS

6.4.5 Coverage of Fault Types ........................................ 156
6.5 Further Analysis of Selected Faults .............................. 157
6.6 Improvement of Test Input Generation ............................ 169
6.7 Further Discussion on Annotation ................................. 173

7 Conclusions and Future Work ........................................ 176
  7.1 Contributions .......................................................... 176
  7.2 Additional Contributions ........................................... 178
  7.3 Limitations .............................................................. 180
  7.4 Future Work ............................................................ 181

A Code of Test Driver Components ..................................... 183
  A.1 The Test-Driver Plan for Testing an Event ...................... 183
  A.2 The Test-Driver Plan for Testing a Belief ...................... 184

B Backus-Naur Form of Input Variable Declaration ................... 186

C Screenshots of The Testing Tool ................................. 189

D Testing Results of The Weather Alert System ....................... 193

E Test Report of The Weather Alert System .......................... 195

F Distribution of Faults .................................................. 200

Bibliography ............................................................... 205
List of Figures

2.1 Plan Selection Mechanism ........................................ 11
2.2 Phases and Artefacts of The Prometheus methodology .................. 15
2.3 PDT Design Overview ............................................. 21
2.4 Structure of A Test Harness ......................................... 36

3.1 Agent Components in Prometheus .................................... 47
3.2 Example of A Plan Cycle ........................................... 53
3.3 A Plan Cycle without Cyclic Execution ............................... 54
3.4 Structure of A Beliefset in JACK (Example) .......................... 57
3.5 Structure of A Beliefset in JADEX (Example) ......................... 58
3.6 Structure of A Belief in JASON or 3APL (Example) ................... 58
3.7 Data Fields of the Book DB Data Entity in Prometheus ............... 59
3.8 Fault Identification for Belief Operations Using JACK Callback Principles (Insert, Update and Delete) .................. 63
3.9 Fault Identification for Belief Operations Using JACK Callback Principles (Modify) ........................................... 64

4.1 Main Components of The Testing Framework .......................... 67
4.2 Global Testing Process for An Agent System ......................... 68
4.3 Example of Cyclic Structure ......................................... 72
4.4 Example of Belief Dependency ........................................ 73
4.5 Example for Determining Order of Testing ............................. 74
4.6 Pseudo Code for Step 2 of Testing Order Determination ............... 75
4.7 Draft Test Queues .................................................... 76
4.8 Test Queues after Removing Plans in Cycles ......................... 77
4.9 Test Queues after Relocating Cycles .................................. 77
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10</td>
<td>Test Queues after Combining Cycles</td>
<td>77</td>
</tr>
<tr>
<td>4.11</td>
<td>Test Queues after Combination</td>
<td>78</td>
</tr>
<tr>
<td>4.12</td>
<td>Test Queues after Adding Triggering Events of Root Plans</td>
<td>78</td>
</tr>
<tr>
<td>4.13</td>
<td>Combination of All Test Queues</td>
<td>79</td>
</tr>
<tr>
<td>4.14</td>
<td>Final Test Queue</td>
<td>79</td>
</tr>
<tr>
<td>4.15</td>
<td>Abstract View of A Test Harness</td>
<td>81</td>
</tr>
<tr>
<td>4.16</td>
<td>Subsystem under Test of A Plan</td>
<td>83</td>
</tr>
<tr>
<td>4.17</td>
<td>Subsystem under Test of A Cyclic Plan</td>
<td>83</td>
</tr>
<tr>
<td>4.18</td>
<td>Subsystem under Test of An Event</td>
<td>84</td>
</tr>
<tr>
<td>4.19</td>
<td>Subsystem under Test of A Belief</td>
<td>84</td>
</tr>
<tr>
<td>4.20</td>
<td>Example of Unit Test Descriptor</td>
<td>90</td>
</tr>
<tr>
<td>4.21</td>
<td>Code of Activation Plan in The Test Agent</td>
<td>92</td>
</tr>
<tr>
<td>4.22</td>
<td>Code of The Test Main Class for The SUT</td>
<td>94</td>
</tr>
<tr>
<td>4.23</td>
<td>Code of Test-Driver Plan for Testing A Plan</td>
<td>95</td>
</tr>
<tr>
<td>4.24</td>
<td>Test Code for Blocking The Execution of One Plan</td>
<td>96</td>
</tr>
<tr>
<td>4.25</td>
<td>Test Code for Reporting The Trigger of The Plan under Test</td>
<td>99</td>
</tr>
<tr>
<td>4.26</td>
<td>Test Harness for a Plan (Detailed Structure)</td>
<td>99</td>
</tr>
<tr>
<td>4.27</td>
<td>Test Code for Reporting Plan Completion</td>
<td>100</td>
</tr>
<tr>
<td>4.28</td>
<td>Test Code for Reporting on The CC Value of The Plan under Test</td>
<td>101</td>
</tr>
<tr>
<td>4.29</td>
<td>Test Code for Reporting on The CC Value of The Plan under Test (Alternative)</td>
<td>101</td>
</tr>
<tr>
<td>4.30</td>
<td>Test Code for Reporting on An Invocation of Event Posting</td>
<td>102</td>
</tr>
<tr>
<td>4.31</td>
<td>Test Code for Reporting on A Plan’s Relevance to An Event</td>
<td>103</td>
</tr>
<tr>
<td>4.32</td>
<td>Test Code for Reporting The Cyclic Execution</td>
<td>104</td>
</tr>
<tr>
<td>4.33</td>
<td>Test Harness for Cyclic Plans (Detailed Structure)</td>
<td>105</td>
</tr>
<tr>
<td>4.34</td>
<td>Example of Cyclic Execution</td>
<td>105</td>
</tr>
<tr>
<td>4.35</td>
<td>Test Code for Checking The Limit of Cyclic Execution</td>
<td>106</td>
</tr>
<tr>
<td>4.36</td>
<td>Test Harness for an Event (Detailed Structure)</td>
<td>107</td>
</tr>
<tr>
<td>4.37</td>
<td>Test Code for Reporting on The Execution of A Plan</td>
<td>107</td>
</tr>
<tr>
<td>4.38</td>
<td>Test Code for Reporting on A Plan’s Relevance to An Event</td>
<td>108</td>
</tr>
<tr>
<td>4.39</td>
<td>Case Study of An Interactee Agent</td>
<td>109</td>
</tr>
<tr>
<td>4.40</td>
<td>Sample Mock Agent Code: StockManager</td>
<td>111</td>
</tr>
<tr>
<td>4.41</td>
<td>Code of the QueryBookbyName plan</td>
<td>112</td>
</tr>
<tr>
<td>4.42</td>
<td>Initial Page of A Test Report</td>
<td>113</td>
</tr>
<tr>
<td>4.43</td>
<td>Example of An Agent Test Report</td>
<td>114</td>
</tr>
<tr>
<td>Figure/Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.44</td>
<td>Example of A Unit Test Report (Overview and Test Inputs)</td>
<td>115</td>
</tr>
<tr>
<td>4.45</td>
<td>Example of A Unit Test Report (Summary and Cases)</td>
<td>116</td>
</tr>
<tr>
<td>5.1</td>
<td>Declaration of Input Variables for A Plan</td>
<td>121</td>
</tr>
<tr>
<td>5.2</td>
<td>Windows of Manual Test Case Inputs</td>
<td>128</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparative Statements</td>
<td>129</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparative Relationships of The First Kind of Comparative Statements</td>
<td>130</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparative Relationships of The Second Kind of Comparative Statements</td>
<td>130</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparative Statements (A Simpler Version)</td>
<td>131</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparative Relationships of Figure 5.6</td>
<td>131</td>
</tr>
<tr>
<td>5.8</td>
<td>Illustration of Simple Variables</td>
<td>135</td>
</tr>
<tr>
<td>5.9</td>
<td>Illustration of Complex Variables</td>
<td>136</td>
</tr>
<tr>
<td>5.10</td>
<td>Value Assignment of Belief Variables</td>
<td>136</td>
</tr>
<tr>
<td>5.11</td>
<td>False Positive Specification</td>
<td>139</td>
</tr>
<tr>
<td>6.1</td>
<td>Structure of The Evaluation Application</td>
<td>144</td>
</tr>
<tr>
<td>6.2</td>
<td>Distribution of Faults</td>
<td>146</td>
</tr>
<tr>
<td>6.3</td>
<td>Example of Fault 2</td>
<td>159</td>
</tr>
<tr>
<td>6.4</td>
<td>Example of Fault 3</td>
<td>161</td>
</tr>
<tr>
<td>6.5</td>
<td>Example of Fault 4</td>
<td>163</td>
</tr>
<tr>
<td>6.6</td>
<td>Code Branches of The Plan $MoveToDestination$</td>
<td>164</td>
</tr>
<tr>
<td>6.7</td>
<td>Code of The Plan $MoveToDestination$</td>
<td>165</td>
</tr>
<tr>
<td>6.8</td>
<td>Belief Variables of The Plan</td>
<td>166</td>
</tr>
<tr>
<td>6.9</td>
<td>Example Two of Plan Failure</td>
<td>167</td>
</tr>
<tr>
<td>6.10</td>
<td>Example Input Variables</td>
<td>170</td>
</tr>
<tr>
<td>6.11</td>
<td>The Comparative Relationships of The Example in Figure 6.10</td>
<td>171</td>
</tr>
<tr>
<td>A.1</td>
<td>Code of Test-Driver Plan for An Event</td>
<td>183</td>
</tr>
<tr>
<td>A.2</td>
<td>Code of Test-Driver Plan for An Event after Code Augmentation</td>
<td>184</td>
</tr>
<tr>
<td>A.3</td>
<td>Code of Test-Driver Plan for A Belief after Code Augmentation</td>
<td>184</td>
</tr>
<tr>
<td>A.4</td>
<td>Code of Test-Driver Plan for A Belief after Code Augmentation</td>
<td>185</td>
</tr>
<tr>
<td>C.1</td>
<td>Accessing Testing Feature from Tools Menu of PDT</td>
<td>189</td>
</tr>
<tr>
<td>C.2</td>
<td>Selection of Implementation and Log Directories</td>
<td>190</td>
</tr>
<tr>
<td>C.3</td>
<td>Testing Window</td>
<td>191</td>
</tr>
</tbody>
</table>
C.4 Starting Test ................................................. 192

E.1 System Level Page of Test Report for WeatherAlert ..................................... 195
E.2 Agent Level Page of Test Report for WeatherAlert ........................................ 196
E.3 Unit Level Page of Test Report for WeatherAlert (Overview and Inputs) ......... 197
E.4 Unit Level Page of Test Report for WeatherAlert (Test Summary) ................. 198
E.5 Unit Level Page of Test Report for WeatherAlert (Details of Cases) .............. 199
List of Tables

3.1 Fault Types of Plans ................................................. 51
3.2 Fault Types of Cyclic Plans ........................................ 53
3.3 Fault Types of Events ............................................... 56
3.4 Fault Types of Beliefs ............................................... 60
3.5 Success-Triggered and Attempt-Triggered Callback Methods in JACK .... 61

4.1 Examples of Initialisation Procedures .................................. 91
4.2 Information Tracked at Runtime for Testable Features ................. 97

5.1 Example of Equivalence Classes ....................................... 125
5.2 List of Value Combinations in The Basic level ....................... 127
5.3 List of value Combinations in The Extended level ..................... 127
5.4 Value Combinations of Figure 5.6 .................................. 132
5.5 Value Combinations Considering Comparisons ..................... 133

6.1 Numbers of Test Cycles .............................................. 155
6.2 Time Cost for Testing A System ...................................... 156
6.3 Fault Types Not Observed as Actual Problems ...................... 157
6.4 Context Conditions of Plans Handling $EAct$ ........................ 169
6.5 Value Set of The Example in Figure 6.10 .......................... 170
6.6 Comparison Set of The Example in Figure 6.10 ................. 171
6.7 Final Value Combination Set of The Example in Figure 6.10 .... 172
6.8 Examples of Annotation Mistakes .................................... 173

D.1 Categories of Faults Detected and the Associated Fault Types of The Weather Alert System ............................................. 194
F.1 Categories of Faults Detected and the Associated Fault Types . . . . . . 204
Abstract

Agent technology has been increasingly used for building sophisticated applications. Agents are autonomous and complex, making their correctness difficult to be verified. Although there has been some work on testing some aspects of an agent system, there is a lack of research for comprehensively testing an agent system automatically from a low-level stage.

This thesis explores an approach for automatically testing agent systems. In the approach an automated testing framework has been developed to allow for completely automated unit testing of an agent system, from generation of test cases, through automated implementation of test harnesses, running of test cases and production of a detailed report. Following the principle of model based testing, the approach tests each unit in the agent system against its details as specified in the unit’s design descriptor, which is part of the AOSE (Agent Oriented Software Engineering) design model of the system. An algorithm has been developed to comprehensively generate test case inputs taking into account value ranges of, and relationships between input variables. Test cases are executed to detect potential faults in a unit under test, based on a fault model we have developed that describes possible problems in different kind of units. Although test cases are generated automatically, we also allow for adding manual test cases with particular values to verify certain situations.

The evaluation results have shown that the tool can reveal previously unidentified problems in a system under test. Some of these problems cause substantial but difficult to detect errors at runtime. Some of the faults detected are false positives, however most of these can be avoided or can come with more precise notifications with the improvements of our testing framework.
Chapter 1

Introduction

This thesis discusses an approach for automatically testing systems that are composed of software agents. A software agent is a software component that carries out autonomous actions to pursue expected goals in a given environment [Wooldridge, 2002, page 15]. There are compelling reasons for the use of this technology. For example, Benfield et al. [2006] has shown a productivity gain of over 300% using BDI (Belief Desire Intention) agents, a common kind of agent, for software development. Agent technology is increasingly being used for building complex software applications in industry [Munroe et al., 2006; Péchouček and Mařík, 2008], such as the application for automated air traffic control [Péchouček and Sislak, 2009]. Padgham and Winikoff [2004, page 16] showed that agent systems provide great flexibility, with over a million ways to achieve a given goal using only a relatively small hierarchy of goals and plans. Because agents are autonomous and flexible agent systems can be difficult to test. Therefore an approach is necessary that can test an agent system effectively and efficiently.

Testing of software systems is critical for assuring quality and for reducing the cost of software problems. A report from NIST (National Institute of Standards and Technology) in the U.S.A has shown that the cost of solving software problems is an estimated $59.5 billion dollars annually and more than a third of the cost could be avoided if better software testing can be carried out [NIST, 2002]. In a survey carried out by SQS (Software Quality Systems) Research covering 250 companies in over 10 countries in 2007, over 80 percent of companies claimed that testing is an important investment 1. However, testing is difficult and time-consuming. Frequently over 50% of the cost of development is spent on software

1http://research.sqs-group.com/archive/market_research_2007.htm
CHAPTER 1. INTRODUCTION

testing [Kit and Finzi, 1995; Hailpern and Santhanam, 2002; Taipale et al., 2005].

The complexity of agent systems and the difficulty of testing software systems raise challenges in testing agent systems. Agent systems consist of components that possess specific characteristics different from those of components of conventional software systems. Agents, which are core components of an agent system, are goal oriented, pursuing their goals even when the environment in which they are situated has changed [Wooldridge, 2002]. An agent usually consists of low-level components, such as plans that achieve particular goals and events that activate plans when particular conditions are satisfied. These agent-specific components behave differently from components in conventional software systems such as methods and classes, leading to potential problems that are specific to agent systems. For example, an event may not activate appropriate plans for goal achievement, or a plan may not post events as expected to achieve sub-goals. Specific characteristics to be verified and potential problems to be revealed in agent systems require specific and deliberate testing strategies.

Testing a software system is a process of revealing inconsistencies between expected behaviour and actual behaviour of the system [IEEE, 1998]. A specification of expected behaviour of a system under test (SUT) can be obtained from the design model of the system. The development of an agent system commonly follows an Agent Oriented Software Engineering (AOSE) methodology, which considers the agent as the central design metaphor [Sycara, 1998]. A design model is usually developed that specifies the features of the system under development in detail, in order to guide the implementation of the system. AOSE methodologies often provide notations of design artifacts that specify structures and properties of components in an agent system. For example, in a design model of Prometheus, which is a well established AOSE methodology [Padgham and Winikoff, 2004], there is an agent descriptor that specifies the properties of an agent, such as the roles played by, the data accessed by and the internal structure of the agent. These details specified in the design are valuable for verifying the correctness of the implementation of the system.

Another issue for testing agent systems is test automation, which is the use of programs to automatically perform some or all activities of a testing process. Test automation has been widely used in testing conventional software systems to increase the effectiveness and efficiency of a testing process [Runeson, 2006], but its usage for agent testing is still limited. Test automation usually requires the development of automatic test cases that can be executed to automatically check existing behaviour of the SUT and to carry out test analysis to reveal possible problems. Automating the testing of agent systems requires a specialised approach as components in agent systems, such as agents or plans, possess specific characteristics that
are different from those of traditional software components. For example, at runtime a plan within an agent may be activated to achieve a goal when an event is posted and a particular condition is satisfied. The plan may post other events to achieve sub-goals. The plan may complete or fail. All such agent specific behaviours need to be observed and analysed to verify the correctness of the plan’s implementation. Since there may be a large number of possible situations that need to be checked for an agent system (e.g. many possible ways for goal achievement), appropriate mechanisms are required that can generate test cases to thoroughly check these various situations.

In this thesis, an approach to automatically test agent systems is presented to address the issues above. The approach tests BDI agent systems, as the BDI structure is a popular and well accepted agent architecture characterised by specifying agent’s mental attitudes of Beliefs, Desires and Intensions [Bratman, 1987; Rao and Georgeff, 1991; 1992]. The approach focuses on unit testing, which is typically the first stage of testing. Units in terms of software testing are the smallest possible testable components in a software system [Burnstein, 2002]. It is recommended that a software system is tested from the stage of unit testing [Myers et al., 2004, page 91], to uncover potential problems in the SUT as early as possible. Unit testing can be of particular benefit in testing of agent systems, as interactions between agent components are more complicated than those in conventional software systems, making it more difficult to identify reasons of faults that are detected when agent components work together. If unit testing has been carried out on low-level agent components such as plans and events, the complexity of testing high-level components such as agents will be significantly reduced. However, conventional approaches for unit testing, such as those for Object Oriented systems [Binder, 1999], cannot be directly used for agent systems due to the specific characteristics of agent components different from those of conventional software components. For example, problems in agent systems may include plans that are not activated by the event as expected, plans that do not complete successfully due to errors in implementation and multiple plans that are applicable for an event when this is not intended. Therefore tailored approaches are necessary for unit testing agent systems.

The approach in this thesis follows the principle of model based testing, which has been widely accepted for automated derivation of test cases [Apfelbaum and Doyle, 1997; Reuys et al., 2005]. The principle is that test cases can be derived in whole or in part from a test model that describes some (or all) aspects of the SUT but is simpler than the SUT. Such a test model can be a design model of the SUT, or a model that has been deliberately developed to describe particular features of the system, for testing purposes. Our approach uses the
CHAPTER 1. INTRODUCTION

design model of the Prometheus methodology that provides design artifacts (descriptors) that specify details of components in the system. With the principle of model based testing, test cases in our approach are automatically generated to thoroughly test components in an agent system.

Our approach has been evaluated using 13 agent systems developed by postgraduate students. Evaluation results have shown that the tool can reveal previously unidentified problems in a system under test. Some of these problems cause substantial but difficult to detect errors at runtime. Some of the faults detected are false positives, however most of these can be avoided or can come with more precise notifications with the improvements of our testing framework.

There is relatively little existing work on testing of agent systems. To our knowledge there is none that systematically and in a fully automated manner does thorough unit testing, although some work such as that of Tiryaki et al. [2007] and Nguyen et al. [2010] does make a start at testing components within agent.

**Research Questions**

Based on the issues associated with agent testing and the limitations of existing work, the following four research questions addressed in this thesis are:

- **What is the scope of unit testing for agent systems?**
  This involves the identification of the basic units in agent systems as units to be tested.

- **How can AOSE design artifacts be used as test criteria?**
  There are two parts to this question. First, what features of an agent system should be verified and what are the potential problems that may exist? Second, what information that is specified in design artifacts can be used to verify system features and to what degree?

- **How can an automated testing framework be developed to effectively capture errors?**
  There are three parts to this question:
  
  - How do dependencies between units/agents affect testing?
    An agent system usually contains multiple agents and an agent contains multiple units. Do the dependencies between these components affect testing?
CHAPTER 1. INTRODUCTION

How is each individual unit in an agent system automatically tested? How are test cases automatically generated and executed and how are test outcomes collected?

How can test outcomes of a unit be analysed to reveal possible problems?

How can a comprehensive set of test cases be generated?

Although testing of most large and complex software systems faces the challenge that there may be a great number of situations to be checked, the hierarchical nature of BDI agent programs exacerbates this issue. In a BDI agent system, a goal can have several choices of plans to achieve it. Each plan can have several steps each of which is a subgoal to achieve, and each subgoal can also has its choices of sub-plans. An example by Padgham and Winikoff [2004, page 16] shows that a goal with 3 levels of hierarchy and less than 100 subgoals may lead to over two million ways to achieve the top level goal. Generally it is impossible to exhaustively test all such ways. Therefore an appropriate strategy must be considered to generate test cases as comprehensively as possible.

Thesis Outline

In the rest of this thesis, chapter 2 introduces background information relevant to this research, including relevant foundational information of multi agent systems and AOSE methodologies. Particularly, the chapter discusses the Prometheus methodology [Padgham and Winikoff, 2004] that is used as the system design model in our research. This chapter also introduces background to unit testing and model based testing and presents existing work on testing agent systems.

Chapter 3 presents the basic types of units identified as units to be tested and the details of the fault model we have defined. In brief, this chapter presents “what to test”.

Chapter 4 discusses how agent systems are unit tested within our testing framework. The chapter presents the process of unit testing an agent system and the algorithm that determines the order of testing for all the units in a system. We have developed a testing tool that follows this process and automatically unit tests an agent system. In the test of each unit, a test harness program is automatically implemented that then generates and executes test cases for testing the particular unit. The chapter presents the details on how this is achieved.

In chapter 5, we present an algorithm for test input generation. Based on this algorithm
the inputs of test cases can be automatically generated considering value domains and relationships between relevant variables. Also, the tester/developer can manually add test cases with particular values if necessary.

Chapter 6 presents details of the experimental evaluation of the testing framework and the results obtained. Finally, conclusions and possible future improvements are presented in chapter 7.
Chapter 2

Background

This chapter reviews relevant literature in both agent systems and software testing to assist in the understanding of the background for this thesis.

Our research explores automated unit testing with AOSE (Agent Oriented Software Engineering) methodologies. This chapter starts with a brief introduction of some concepts, characteristics and an architecture of agents in Section 2.1, and then in Section 2.2 discusses general features of AOSE methodologies. In Section 2.3 the Prometheus methodology is introduced, which is used in our research as the system design model based on which test cases are generated. The following section describes some principles of software testing that are related to our research, including unit testing, test automation and model based testing. Finally, the last section reviews existing work in testing of agent systems both automatically and semi-automatically.

2.1 Agents

The term agent in this thesis refers to the concept of a software agent that is a computer program that carries out tasks on behalf of other software agents or humans. Given the popularity of agent research, there are many agent definitions given by different researchers [Russell and Norvig, 1995; Maes, 1995; Smith et al., 1994; Hayes-Roth, 1995; Wooldridge and Jennings, 1995; Franklin and Graesser, 1997]. Most of them agree with Wooldridge [2002, page 15] that:

"An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives. "

(October 14, 2011)
Most researchers describe characteristics of an agent as follows:

1. autonomous: agents are independent and make their own decisions, without direct interventions from other agents, or humans.

2. reactive: agents react to changes that happen in the environment in which they are situated.

3. pro-active: agents act in a goal-directed manner. An agent is able to pursue their own goals even when the environment has changed.

4. social: agents interact with other agents via some kind of communication protocol.

In addition, in some specific agent architectures, such as the Belief Desire Intention (BDI) architecture [Bratman, 1987; Rao and Georgeff, 1991; 1992], extra characteristics of agents are desirable:

1. robust and flexible: agents are able to recover from the failures of their actions or plans due to environment changes, and choose new actions to achieve their initial goals. To achieve this robustness, agents are flexible with the range of methods that can achieve a goal.

2. situated: agents may be situated in an environment that is dynamic, unpredictable and unreliable. Dynamic indicates that the environment may not remain static while agents are trying to achieve their goals. Unpredictable reflects the impossibility of agents to predict future states of the environment. The environment is unreliable in that agents may fail to achieve their goals for reasons that are beyond their control. For example, a delivery agent may fail to deliver a package to a customer because the customer’s address has been changed.

An agent architecture is an abstract representation of philosophical models that are proposed for explaining rational behaviours of intelligent agents. One popular agent architecture is the Belief Desire Intention (BDI) architecture, which characterises agents by specifying their mental attitudes of beliefs, desires and intentions. Agents discussed in this thesis are based on the BDI architecture.

There are other kinds of agent architectures such as SOAR [Laird et al., 1987] and RAP [Firby, 1987]. However, this thesis focuses on BDI agents.
Belief Desire Intention Agents

The BDI model is based on the philosophical work of Bratman [1987] on the reasoning of rational agents. This was formalised by Rao and Georgeff [1991; 1992]. In the BDI model, an agent’s mental state consists of beliefs, desires and intentions, which are explained below:

- **Beliefs** correspond to the information an agent has about its states and the environment in which it is situated. An agent’s beliefs are believed to be true by the agent itself, but may not actually be. That is why the term “belief” is used as opposed to “knowledge” or “fact”. For example, a *Travelling* agent’s belief that taking a bus is the cheapest means of travelling from Sydney to Melbourne may not be true at weekends when the railway company gives a half-price discount (the environment has changed).

- **Desires** are the states that the agent wants to reach, or what an agent would wish to satisfy. Rao and Georgeff [1992] developed BDI theories with goals rather than desires. The difference between goals and desires is that goals should be consistent and believed to be possible by the agent, while desires can be inconsistent. For example, a *Travelling* agent cannot simultaneously pursue a goal to go to Melbourne and another goal to go to Sydney.

- **Intentions** reflect the decisions an agent makes regarding how to attain its desires, or the means for achieving its goals. For example, a *Travelling* agent has a goal to leave for Sydney from Melbourne. To achieve this goal, it will intend to look for an appropriate means of getting there. The agent may review different ways of travelling such as by train, aeroplane or car, according to its belief of how much each means will cost and how long it will take to travel. It will then adopt an intention to use one of these means.

The architecture of a BDI agent has been formalised for BDI based reasoning [Cohen and Levesque, 1990; Rao and Georgeff, 1991; Singh and Asher, 1991]. Rao and Georgeff [1992; 1995] have proposed a BDI interpreter that explains an execution cycle to show how the BDI architecture is used to achieve agent behaviours in a dynamic environment. There have also been a series of agent implementation platforms based on the BDI model, such as PRS [Ingrand et al., 1992], dMARS [d’Inverno et al., 1998], JAM [Huber, 1999], JACK [Winikoff, 2005], JADEX [Pokahr et al., 2003] and JASON [Bordini et al., 2007].
Plans and Events

In most BDI based implementation platforms, the Belief Desire Intention model and the original execution cycle of BDI agent behaviours have been adapted to suit a practical dynamic environment. Intentions are often realised as plans, which are the means used by or the options available to the agent for achieving certain goals [Rao and Georgeff, 1995]. Also goals are presented as events (or triggers [Winikoff, 2005], or motivations [Jennings, 1995]) that trigger plans. A commonly accepted plan structure consists of a name, a body, an invocation condition and a context condition. A plan body describes a set of actions to be performed and subgoals that have to be achieved for the successful execution of the plan. The invocation condition of a plan specifies if the plan is relevant to handle a particular event. A plan’s context condition, also known as pre-condition [Rao and Georgeff, 1992; Wooldridge, 2000], specifies the situation in which this plan can be used.

A typical plan selection mechanism triggered by an event in a BDI agent is shown in Figure 2.1. An agent has a collection of plans that represents an agent’s options in response to events. An agent selects from its plan collection a set of plans $P_1 \sim P_m$ in response to an event, which is either generated internally or from the external environment. The selected
plan set consists of all plans with a matching invocation condition. Then, by considering context conditions of the plans in the selected set, the agent determines a sub-set of applicable plans $P_i \sim P_k$ as candidates for execution. From these candidates, one plan $P_j$ is selected for execution. The choosing strategy can depend on plan priority or can be random, though an alternative plan in the applicable set may be selected if the chosen plan fails. The execution of a plan may result in actions interacting with the environment and events that trigger other plans for achieving subgoals. In addition, during the execution of a plan, the agent may also receive new events from the environment, indicating changes in the environment.

2.2 AOSE Methodologies

Agent based software development approaches are appropriate for developing software applications when the environment is open, complex or uncertain [Jennings and Wooldridge, 1998; Munroe et al., 2006; Pechouček and Mařík, 2008]. They are also suitable for the development of systems that involve distribution of data, control, expertise or resources, and for situations under which the system is naturally regarded as a society of autonomous cooperating components, for example, agents as “expert assistants” [Maes, 1994; Jennings and Wooldridge, 1998] in an expert system, agents as humans in affective social games [Conati and Zhao, 2004; Sollenberger and Singh, 2009] and agents as residents in a city in 3D virtual worlds [Bogdanovych et al., 2011].

The development of agent based software systems requires appropriate software engineering methodologies, which can construct software applications with agents as the central design metaphor. There are a series of Agent Oriented Software Engineering (AOSE) methodologies that have been developed over the last decade. In general, they have two main functionalities: enable developers to systematically build multiagent applications and provide industrial-strength toolkits that are flexible enough to specify numerous characteristics of agents [Sycara, 1998]. Most AOSE methodologies borrow ideas from existing approaches and methodologies, such as some Object Oriented (OO) methodologies. Padgham and Winikoff present Prometheus as an agent oriented methodology based on BDI agents [2004]. The methodology shares similar iterative development process with the RUP (Rational Unified Process) [Kruchten, 1998] and utilises some UML (Unified Modeling Language) diagrams where they are compatible with the agent oriented paradigm. Gaia is a general purpose methodology [Wooldridge et al., 2000; Zambonelli et al., 2003] and borrows notations from the OO methodology of Fusion [Coleman et al., 1994] for modelling work. The Tropos
methodology [Bresciani et al., 2004] is a requirement-driven methodology that follows the concepts and notions of the i* modelling framework [Yu, 1996], with a special focus on modelling requirements compared to other methodologies. The MaSE (MultiAgent Systems Engineering) methodology was originally presented for applications of general-purpose multiagent systems [Wood and DeLoach, 2001] and then was extended as O-MaSE [DeLoach, 2005; DeLoach and Garcia-Ojeda, 2010], which supports organisation-based development. O-MaSE constructs the system as a multiagent organisation, with respect to the interactions with the environment and the capabilities of roles. The PASSI (Process for Agent Societies Specification and Implementation) methodology [Burrafato and Cossentino, 2002; Cabrera-Paniagua and Cubillos, 2009] integrates design models and ideas from Object Oriented Software Engineering (OOSE) and multiagent systems and uses UML as its main notation. ASPECS is another organisation-based methodology that extends UML notifications to present agent concepts such as goals, agents and capabilities. The methodology supports the development of Holonic agents, each of which can be composed of other holonic agents as sub-structures [Cossentino et al., 2010].

**Design Artifacts**

AOSE methodologies in general specify different development phases to guide system development. Design artifacts are produced in development phases for capturing the information of the system structure. Design artifacts encode system requirements in such a way that the system can be implemented to realise expectations. They are represented in various forms from plain textual description to graphical notations. Some of them are intermediate artifacts and some are final ones. The final design artifacts are generally concrete and well structured and provide detailed understanding for implementation. The artifacts can be considered as partial specification of correct system behaviour. For example, some artifacts that are commonly specified in AOSE methodologies are:

In the requirements analysis phase -

- **actor (or role)** artifacts are defined to represent external stakeholders that interact with the system and to help developers understand how the system interacts with the environment.

- **system goal** artifacts are specified to capture the requirements of the system and to facilitate the understanding of the reason for building the system. Commonly a goal
hierarchy is defined by decomposing goals into subgoals.

In the design phases -

- a *system overview (or agent model)* artifact is defined to represent the overall structure of the system in terms of agents.

- *protocol* artifacts are defined to represent the interaction between agents. A protocol contains a structure of messages between two agents and may also specify the interactions of an agent with the environment.

- *agent descriptor (or agent class)* artifacts are defined to specify properties of agents, such as the roles played, the data accessed and the capabilities of agents.

- *plan (or plan descriptor)* artifacts are defined to represent low-level behaviours of agents. For example, the Prometheus methodology defines a *plan descriptor* artifact that specifies internal details of a plan, such as the textual description of the plan’s body, the conditions under which the plan may be applied and the subgoals triggered by the plan.

Design artifacts produced during the development form the system design model that is the basis for the understanding of implementation. The information contained in design artifacts guides the implementation work. In the research presented in this thesis, design artifacts are utilised for verification of implemented agent systems.

The Prometheus methodology [Padgham and Winikoff, 2004] is a well established agent development methodology that specifies the production of design artifacts through the analysis and design phases. These artifacts represent details of basic agent entities such as goals, events, plans and beliefs. The research upon which this thesis is based explores a model based framework for unit testing in agent systems, where the models used are provided by the design artifacts of the Prometheus methodology. Prometheus will be introduced in some detail in the next section before the literature on relevant software testing concepts is summarised.

### 2.3 Prometheus

The Prometheus methodology is an agent oriented methodology for developing software systems based on the BDI agent architecture. It has been developed for over ten years and has been continually refined and improved though teaching activities and industry seminars.
Prometheus defines a detailed development process covering the phases of system specification, design, implementation and testing/debugging. In addition, it defines a range of graphical or textual artifacts that directly relate to agent concepts and facilitate the tool support for the methodology. The methodology specifies in detail three main phases for the design of an agent system: \textit{system specification, architectural design} and \textit{detailed design} (Figure 2.2). The implementation details depend on the platform that is chosen.

\textbf{System Specification}

The \textit{system specification} phase (or \textit{Analysis}) constructs requirements to form a detailed and documented understanding of the system. \textit{System specification} starts with the identification of external entities (or \textit{actors}) that will interact with the system. For example, in a Conference Management System (CMS) (a case study of Prometheus [Padgham et al., 2008b]), four actors are identified: an Author, a PCChair, a PCMember and a Reviewer. The identified \textit{actors} are then associated with \textit{scenarios}, each of which describes a set of particular interactions.
between actors and the system, identifying inputs to and outputs from the system. For example, the Author is associated with the *get papers* scenario, which describes how papers are submitted by an author to the system, while the Reviewer and the PCMember actors are associated with the *review* scenario, which describes the process of how a paper is assigned by PC members to reviewers for reviewing. During the identification of *actors* and *scenarios*, the input to and output from the system are also specified. The former are termed *percepts*, such as a paper submitted to the *get papers* scenario. The latter are termed *actions*, such as an acknowledgement given by the *get papers* scenario back to the author who has submitted a paper.

The next step is to specify the detail of each scenario in order to understand the main functionality of the system. A scenario is constructed as a series of steps which describes how it is performed. Each step is in the form of a goal, an action, a percept, or a sub-scenario. For example, a *review* scenario can consist of the steps of **goal**(invite reviewers), **goal**(collect preferences), **goal**(assign papers to reviewers), **action**(send papers to reviewers), **percept**(receive reviews from reviewers) and **goal**(collect all reviews). The *roles* that are associated with each step may also be indicated, as well as the data accessed.

During the specification of scenarios, some of the sub-goals are also identified. The use of goals at the requirement engineering and system specification phases facilitates not only the understanding of the reason for building the system, but also a mapping of goals into the products generated in later detailed design and implementation phases. Often one goal is created for each scenario and when a scenario is decomposed into steps, goals in lower levels are also specified. The hierarchy of goals is also created by analysing each goal and identifying its subgoals using an AND/OR decomposition. The AND decomposition results in a series of smaller subgoals that are parts of achieving the high-level goal. The OR decomposition defines subgoals that are alternative approaches for achieving the goal. For example, the **goal**(invite reviewers) may be decomposed using the AND operation to three subgoals: **goal**(confirm reviewer candidates), **goal**(contact reviewers) and **goal**(confirm acceptances). While the **goal**(contact reviewers) may be decomposed using the OR operation to two alternative goals: **goal**(contact by email) and **goal**(contact by phone).

There may be multiple iterations between scenario development and goal identification until the developer believes that the application is sufficiently described. During this process, goals are grouped in such a manner that a group of goals can be assigned to the same system *role*, which is intended as a small and easily specified chunk of system functionality. For example, in the *review* scenario, the goals of “assign papers to reviewers” and “send
papers to reviewers” are both related to an assignment task and can be grouped together and assigned to a system role called “Assignment”.

The products of system specification provide the basis of the following design phases. The scenario model and the goal/role model can be directly used in the following architectural design phase.

**Architectural Design**

The aim of the architectural design phase is to establish the system structure in terms of agents based on the products of the system specification phase. There are two main tasks in this phase: decision of agent types and definition of agent interaction.

The decision of agent types is based on the role model specified in the system specification phase. Roles, incorporated with the goals performed by them, are grouped into agents according to various criteria such as the relationship between roles and the data they access. The roles that seem to be related or share the same information are recommended to be grouped together. For example, in the role model of the CMS, there is the “Assignment” role that performs the goals of “assign papers to reviewers” and “send papers to reviewers”, and the “Review Management” role that performs other reviewing related goals such as “invite reviewers” and “collect reviews”. These two roles can be grouped together as a Review Manager agent because they are both related to reviewing tasks. Each agent type is specified as an agent descriptor that contains information about the agent type, such as its name, its initialisation and demise and a list of goals it is responsible for.

The decision of agent types, including the specification of roles in the system specification phases, strongly depends upon the perspective of designers and upon the application. For the CMS example, the designer could also directly define a “Review Management” role instead of the “Assignment” and “Review Management” roles and then map it to an agent of the same name. At one extreme, every role could be mapped to one agent, and at another extreme, all roles could be grouped to only one agent. Prometheus only provides guidelines for these design operations. The quality of the design still depends on the designer’s opinions and the application requirements.

The interactions between agents are specified in the form of protocols according to the scenario model, which has described the process aspect of the system. Steps in each scenario are analysed to derive interactions between agents, consisting of a specification of allowable sequences of messages. Each message is sent from one agent to another. A protocol in
Prometheus may also include percepts and actions that interact with the environment.

The result of the architectural design phase is a System Overview diagram which describes the structure of the system in terms of agents as well as the protocol definitions.

Detailed Design

The detailed design phase of Prometheus is concerned with the construction of agent internals, with respect to how agents achieve their goals. Each individual agent is refined to specify its internal modules, called capabilities, according to the related goals to be achieved. Often capabilities correspond to the roles assigned to the agent. The internal structure of a capability may contain lower level capabilities if necessary and contain plans and events that are constructed for achieving goals. The internal structure of every agent is outlined in its associated Agent Overview Diagram and the structure of every capability is outlined in its associated Capability Overview Diagram. Within an agent or a capability, plans are defined for achieving the goals of the agent, as well as the events that trigger these plans and shared data entities that are accessed by these plans.

A plan in Prometheus is constructed following the essential structure of a BDI plan, which has been introduced in Section 2.1. A plan must respond to the relevant event and be suitable for a particular situation, which is specified in the context condition of the plan. The absence of the context condition of a plan indicates that the plan is always applicable for its triggering event. A plan’s body specifies the sequence of steps performed to achieve the goal denoted by its triggering event. In some steps internal events may be posted in order to trigger subtasks, or external messages may be sent to communicate with other agents.

Events trigger the selection and execution of plans. Events in Prometheus are internal events within an agent, external messages between agents and percepts from the environment. Important design issues about events are the features of coverage and overlap, as well as the information carried in the event. When an event is posted, some of the plans that handle it will be applicable for the event if they are triggered and their context conditions are evaluated as true. An event may have one or more applicable plans under some circumstances and have no applicable plan under some others. The terms coverage and overlap are introduced within Prometheus to describe those circumstances. An event has complete coverage if it has always at least one applicable plan in any situation. Otherwise, the event has incomplete coverage. Particularly, complete coverage of an event can be guaranteed if the context conditions of the plans that handle the event cover the complete value domains of the environmental variables.
participating in those context conditions. For example, an event has complete coverage if two plans are specified as handling it, one with the context condition “temperature > 0” and the other with “temperature ≤ 0”. This is because the context conditions of these two plans have covered all possible values of the temperature \(^1\). In addition, in some situations the event may have more than one applicable plan. Such situations are termed overlap in Prometheus.

Shared data accessed by plans is specified in detailed design as data entities, such as a Customer DB entity that stores the information of customers. The representation of a data entity can be any data structure and depends on implementation platforms. For example, one common kind is a structure that contains records, such as beliefsets in JACK [Winikoff, 2005].

A data entity may also contain a set of action rules, each of which specifies a particular action activated when the data entity changes. For example, an action rule can specified that when a new customer is inserted into the Customer DB entity, a notification should be given. In Prometheus, an action is denoted by an event that is posted out by the data entity to trigger relevant plans. An action rule is specified as a tuple \(<operation, event>\). Operation can be one of insert, delete, update or modify. For example, an action rule can be specified as \(<\text{Insert, Notification}_\text{Event}>\), indicating that a notification event will be posted out if an insertion occurs to the Customer DB entity.

Tool Support

The Prometheus Design Tool (PDT) [Padgham et al., 2008a] is a freely available tool that supports the development paradigm of Prometheus. It provides users with support to develop the design model of the system. In general, PDT allows the user to:

- construct the graphical model of the system structure
  
  In PDT, the user can create overview diagrams of the system, agents and capabilities, by organising the positions and relations of graphical entities that denote various agent concepts. For example, the user can connect a message entity to a plan entity using an arrow line, with its arrow pointing to the plan. This specifies that the message is handled by the plan.

- develop the process description for scenarios and protocols
  
  PDT provides graphical interfaces for the user to specify the steps of each scenario and the interactions in each protocol in the AUML [Odell et al., 2001] format.

\(^1\)assuming that the temperature is always known
generate hierarchical views of the agents architecture
By using capabilities and capability overview diagrams, each agent can be developed with several layers as needed in order to keep each layer manageable in size.

specify detailed design descriptors for entities
In PDT, the descriptor of each entity contains a mixture of free text fields and structured fields. The user can edit these fields to specify a particular entity, such as an agent or a plan.

test consistency between models
PDT provides the functionality to automatically check consistencies between different models. For example, every goal should be associated with a particular agent and every message should be sent and received by associated plans.

generate a design document
PDT can generate a customisable HTML design document that contains both figures and textual information of all design entities, as well as printable images of various diagrams.

and generate skeleton code for implementation
PDT is able to generate skeleton code of the JACK language [Winikoff, 2005], based on which the developer can continue implementation work to complete the development of the whole system. The developer can iterate between design and implementation.

Figure 2.3 shows the PDT design overview of the CMS example.

In PDT, the details of each entity, such as a plan or an event, is specified in the corresponding detailed design descriptor. A detailed design descriptor represents the details of the relevant entity that are specified in the design. Based on such details the entity will be implemented. For instance, in the design descriptor of a plan, some important properties are specified as follows:

- the name of the plan
- the textual description of the context condition, such as “The budget is more than 2000 dollars.” for a Buy Air-ticket plan
- the triggering event of the plan
CHAPTER 2. BACKGROUND

Figure 2.3: PDT Design Overview

- the outgoing events, which are internal events posted out by the plan to trigger subtasks, or external messages sent out by the plan to communicate with other agents
- the percepts handled by the plan from the environment
- the actions performed by the plan in the environment
- the data entities accessed by the plan
- the textual description of the procedure performed in the plan body

Similarly, the design descriptor of an event gives the details of:

- the name of the event
The details specified in the design descriptor of a data entity include:

- the name of the data entity
- the data type
- other entities that access the data entity
- data fields
- action rules

There are also design descriptors for capabilities and agents. For more details about design descriptors, please refer to [Padgham and Winikoff, 2004] and [Padgham et al., 2008a].

PDT has been used by others to produce code of Jadex [Sudeikat et al., 2005] and 3APL [Jayatilleke, 2007]. The CAFnE toolkit [Jayatilleke et al., 2006] extends PDT by requiring more detailed model based specification and can automatically produce executable code. There has also been a tool for runtime debugging based on PDT models [Poutakidis et al., 2002; Padgham et al., 2005].

2.4 Software Testing

Development of software systems, including agent systems, usually includes several stages that are requirements analysis and definition, design, implementation, testing and system deployment [Pfleeger and Atlee, 2008, page 24]. The stage of testing confirms that the System under Test (SUT) behaves as expected and is important to guarantee the quality of a developed system. This section introduces some background knowledge on testing and the testing techniques that are relevant to the research in this thesis.

Testing is concerned with establishing that a developed system functions properly taking into account the requirements of the user and the associated system specification. These two
essential aspects of testing are commonly referred to as validation and verification [Burnstein, 2002, page 6; Patton, 2005, page 48]. Validation is associated with the question “Have we developed the right software?”, while verification asks “Have we developed the software right?” The commonly acceptable definitions of these two terms, by IEEE Standard Glossary of Software Engineering Terminology [IEEE, 1990], are

“Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements.”

“Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.”

Both aspects are critical to the assurance of software quality. The research in this thesis is concentrated on the verification aspect. Work on validation is usually relevant to requirements engineering [Nuseibeh and Easterbrook, 2000], which focuses on the approaches for precisely specifying requirements.

2.4.1 An Introduction to Testing

Although there are some different opinions about the precise definition of testing, it is commonly accepted that testing is:

“The process of analysing a software item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of the software item.” [IEEE, 1993b]

There are also well established definitions of testing from other researchers:

“Testing is the process of executing a program with the intent of finding errors.” [Myers et al., 2004, page 6]

“Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected

In general, software testing is the process that detects software defects in the SUT so that software faults (or bugs) may be discovered and removed. Ideally, a mature software testing approach should be effective at finding defects which are there with as many details as possible and also be efficient, performing the test process as quickly and cheaply as possible [Fewster and Graham, 1999, page 3].

Software Faults

Faults, or the similar term bugs, describe defects of a software program. There are also other similar terms, such as mistakes and failures, which are commonly related to software defects. IEEE Standard Glossary of Software Engineering Terminology [1990] defines these terms as:

1. mistake: “A human action that produces an incorrect result”
2. fault: “An incorrect step, process, or data definition in a computer program”
3. failure: “The inability of a system or component to perform its required functions within specified performance requirements”. A failure is usually a “result of a fault”.
4. bug: common term for a fault.

The differences between these concepts can be explained by the following example: A requirement of a bookstore system specifies that the user can purchase at most four books in an order. During the development process, a programmer makes a mistake by incorrectly coding the purchase limitation of an order as three. That mistake causes a fault (or a bug) that the program incorrectly gives a warning message when an order with four books is made by the user. This fault reflects a software failure since the program has no ability to fulfill any order with four books.

Bugs and faults are the most common terms used to describe software defects revealed by a testing process. In this thesis the term fault is used.

Software faults may be caused by various circumstances. For example, testers may have insufficient background knowledge to correctly develop the functionalities required; poor communication between developers and users may result in misunderstanding requirements and may lead to incorrect development; developers may neglect some requirements specified in the design and forget to implement them; developers may understand the requirements...
but make mistakes when implementing them. Burnstein has summarised in his book the kinds of circumstances under which faults may occur [Burnstein, 2002, page 39].

It is widely agreed that faults should be found as early as possible, because the cost of fixing them is dramatically increased over time as the development progresses. Patton [Patton, 2005, page 18] emphasises the importance of detecting faults early by denoting an uncovered fault as a bug that grows logarithmically throughout the development process. There is also evidence from other researchers supporting the view that the later faults are detected, the more expensive the cost of fixing them is [Boehm, 1981; Myers et al., 2004; Mcconnell, 2004].

Fault Model

An effective way for revealing faults is to define for a SUT a fault model, which specifies the assumptions about under what situation a fault is likely to be found in the SUT [Binder, 1999, page 51; Myers et al., 2004; Burnstein, 2002, page 42]. Each assumption introduces the occurrence of a software failure and such an occurrence in the SUT can be identified as a fault that exists in the system. For example, in the conference system discussed in section 2.3, a paper can only be assigned to three reviewers at most. So an assumption can be defined like the following:

\[ \text{It is a fault if the Review Manager agent assigns to a reviewer a paper that is already assigned to three persons.} \]

Such an assumption is similar to how a doctor diagnoses patients according to the knowledge of possible diseases and symptoms. Testers then follow this assumption to check if relevant faults exist in the SUT. A practical assumption may be more complicated containing a structured description of failure occurrences and fault identification. During the testing process for the SUT, faults can be revealed according to such assumptions. This strategy of software testing is commonly known as fault directed testing [Binder, 1999, page 66].

By comparison, another strategy is conformance directed testing, which intends to demonstrate the conformance to the requirements or the specifications of the SUT [Binder, 1999, page 66]. Conformance directed testing need not consider assumptions of failure occurrences. Instead, it is concerned with representing features of the SUT as fully as possible to establish conformance to the required capabilities. Conformance directed testing is more application dependent than fault directed testing. However, goals of these two are not mutually exclusive.
Most testing technologies achieve both. This thesis is more concerned with fault directed testing and the testing approach explored in this thesis is application independent.

Levels of Testing

In general, the testing process of a system is applied in different levels taking into account the scope of the parts of the system to be tested and their complexities and maturities [Abran et al., 2004]. The testing process in general begins with unit testing (or module testing), which tests the smallest units or components to identify their functional and structural faults if they exist. After all components are tested and necessary repairs are made, they are integrated into sub-systems and integration testing is conducted to find any fault of unit interactions. System testing starts when all components have been successfully integrated and tested. It focuses on evaluating quality-related requirements, such as performance, usability and reliability. Then, acceptance testing verifies if the system meets all customer requirements. This often consists of alpha testing and beta testing. The former asks potential customers or other teams to test the system at the developers’ site. The latter releases a beta version of the system and asks potential customers to use the system under real-world conditions, and report any faults they found.

2.4.2 Unit Testing

Unit testing, as mentioned in the last section, reveals faults of individual units in early stages of the development life cycle. It uses the basic principle that testing is at first concentrated on the smaller building blocks of a program rather than testing the program as a whole. Following the principle of unit testing, each unit in the SUT is tested separately to verify if it works as expected before it is integrated with other units as modules.

An Introduction to Test Unit

Before further discussion on unit testing, it is necessary to understand what a test unit is within the paradigm of software testing. There are similar definitions of a test unit from different researchers, such as “the smallest executable unit” [Binder, 1999, page 49], “the smallest possible testable software component” [Burnstein, 2002, page 137], “an individual portion of code, a component” [Dustin, 2002, page 143], “individual subprograms, subroutines, or procedures in a program” [Myers et al., 2004, page 91], “software code at its smallest functional point” [Huizinga and Kolawa, 2007, page 49] and “the smallest piece of testable
CHAPTER 2. BACKGROUND

In general, a test unit is one of the smallest building blocks that can be tested in a SUT.

However, the identification of test units in a SUT varies in different types of software systems, depending on the degree to which the source code is encapsulated as blocks and on the perspective of developers. A test unit in a procedure-oriented system is usually a procedure or a function that performs simple tasks. In Object-Oriented (OO) systems, classes are usually suggested as test units because they are mainly encapsulated units in OO systems. In agent oriented systems, there is some speculation on the identification of test units. Some researchers identify agents as test units when they study approaches for unit testing agent oriented systems [Knublauch, 2002; Rouff, 2002; Caire et al., 2004; Seo et al., 2004; Zheng and Alagar, 2005; Coelho et al., 2007]. While some others [Nguyen et al., 2008a; Tiryaki et al., 2007; Nguyen et al., 2008b] explore the verification of internal activities within an agent, such as the goal hierarchy or the plan hierarchy of an agent. They identify internal components of an agent such as goals and plans, as test units. The research in this thesis analyses components within an agent and their relationships and identifies test units within an agent, such as plans, events and beliefs. Chapter 3 will discuss in detail our approach for the identification of test units.

Identification of test units is not restricted to the discussion above and is somewhat flexible in practice, taking different encapsulation levels of units into consideration. For example, in OO systems, not only classes, but also member methods of a class can be identified as test units. This occurs when developers intend to verify system behaviour and system states at both the class level and the method level. Similarly, when an agent oriented system is tested, components in low encapsulation levels, such as member methods or classes, can also be identified as test units. These low level units are often tested in the first place before the test of agent oriented units, such as plans, events and agents, which are at higher encapsulation levels.

The necessity of performing unit testing for software systems has been introduced by some researchers [Dustin, 2002, page 143; Vaaraniemi, 2003; Burke and Coyner, 2003, page 18; Myers et al., 2004, page 91; Huizinga and Kolawa, 2007, page 75] as follows:

1. Unit testing is a way of discovering software defects as early as possible, so the cost of detecting and removing defects is much less compared to other later stages of testing.

2. By testing all individual units of the source code, a unit testing process for a SUT

\[\text{software in the application}\] \footnote{Microsoft Unit Testing: \text{http://msdn.microsoft.com/en-us/library/aa292197(VS.71).aspx}}
CHAPTER 2. BACKGROUND

increases the confidence that the code works correctly without errors.

3. It enables testing parts of a system without waiting for the availability of other parts.

4. Parallelism in testing is able to be performed as many developers can test different units in a SUT and fix problems simultaneously.

5. Unit testing enables a high level of structural coverage of the code. For example, unit test cases can be designed to cover branches of the source code, or conditions in the expressions in the source code as comprehensively as possible.

6. It enables testing internal conditions of the SUT, such as conditions in which exceptions are thrown. Those conditions are difficult to reach by using external inputs of the whole system.

7. Debugging work is facilitated by limiting debugging to a small unit in which faults are searched for.

Unit Test Cases

The practice of unit testing a software system in general requires the generation and execution of appropriate test cases, each of which verifies a particular system feature, such as a system behaviour or state, in some way. This thesis discusses test cases within the scope of unit testing, though test cases are commonly used for all levels of testing.

A test case for testing a unit typically includes four parts [IEEE, 1998] as follows:

1. The identifier of the unit to be tested, such as the name of a class or an agent to be tested

2. A set of test input values received by the unit to be tested

3. A sequence of testing steps which will be manually or automatically applied to the unit in order that a particular feature of the unit can be verified. For instance, these steps can specify how to partially or completely execute the SUT in order to observe behaviour or states of the unit. They can also be the steps to examine the implemented code in order to reveal potential mistakes.

4. The predicted outcomes of the test case

Predicted outcomes describe what should occur after the steps mentioned above are
performed, such as things that should be created, changed, updated or deleted, or things that should not be changed. These predicted outcomes are used to verify if the unit has the particular feature as expected.

The form of a test case can vary depending on the application. For instance, a test case may be a specification that guides human testers for performing the steps specified in the test case, such as a paragraph of textual description in a document, or a structured description in some testing tools. It may also be a block of code that can be executed to receive input, perform steps and output some results.

For example, in a conference system, a Reviewer agent can only review three papers at the most. The agent is expected to reply with a rejection message if more than three papers are allocated to it. Then a test case can be designed to verify that the agent behaves as expected. The four parts of this test case are:

1. The name of the agent to be tested: Reviewer
2. The input of this test case is a set of four papers, which are included in a message that will be sent to the Reviewer agent.
3. The sequence of steps for verification is specified as:
   - Step 1: create an instance of the Reviewer agent
   - Step 2: send a message with the input data to the agent instance
   - Step 3: observe the response of the agent instance
   - Step 4: compare the response with the predicted outcome and report the result
4. The predicted outcome is: the agent should respond with a rejection message reporting that the number of papers (four) received is over the allowed number of three.

This test case can be developed as a program that performs the steps and analyses outcomes automatically. A fault will be detected and reported by the program if the actual response is different from the predicted outcome. For example, the reply message from the agent instance shows that all papers are accepted, or there is no reply from the agent instance at all. Otherwise, if the agent sends out a rejection message as expected, the test case will pass.

Test cases in practice may describe more comprehensive information than the four essential parts mentioned above. For example, special environmental needs may be specified in
a test case. They may include particular runtime environmental states that are necessary for the execution of the SUT, or auxiliary code developed for the purpose of testing. How detailed a test case is depends on the application and on the strategy for designing test cases.

2.5 State of The Art

This section discusses existing strategies and approaches for software testing, especially for unit testing and testing of agent systems that are related to the research presented in this thesis. There have been various strategies developed for unit testing a software system and they usually follow similar stages as presented in section 2.5.1. In order to improve the efficiency of a software testing process, the whole process or parts of the process is often automated. Existing approaches of test automation are discussed in section 2.5.2. The following section 2.5.3 discusses model based testing that is a popularly accepted strategy for derivation of test cases. With those strategies and approaches applied, there have been a couple of approaches developed for testing agent systems as discussed in section 2.5.4.

2.5.1 Unit Testing Strategies

Although various strategies have been developed for unit testing a software system [Burnstein, 2002, page 143; Burnstein, 2002, page 138; Myers et al., 2004, page 91; Perry, 2006, page 83; Juristo et al., 2006], they usually follow three main stages: perform the test planning, acquire the test set and measure the test unit [IEEE, 1993a]. In the first stage features to be tested of the units in the system are identified and the resources and schedule required are estimated. In the second stage a set of test cases is designed and implemented for verifying identified features of each unit. Test procedures are also specified and particular environmental needs are implemented. In the third stage, following the test procedures that have been specified, test cases are executed for testing each unit and test outcomes are evaluated. The details of these three stages are discussed as follows:

• perform the test planning

In this stage the features of the SUT, such as functional requirements, constraints, states and state transitions, and control structures, are studied and those that are relevant to test units and need to be verified are identified. For the instance of the Reviewer agent discussed above, a constraint specified in the design is that an instance of the Reviewer agent only accepts three papers at most. Thus the limitation of three
papers is a feature to be verified. In addition, test features can also be identified based on general characteristics of a particular type of test unit. For example, *Inheritance* is an important feature of Object Oriented programs. Binder [1999, page 69, page 501] discusses *inheritance* as a test feature of an Object Oriented program in his book and introduces the potential faults that are caused by incorrect implementation of *inheritance*.

Based on the test features identified, a fault model of the SUT is defined which consists of assumptions about what faults are likely to be found in the SUT [1999, page 51, page 334]. Each assumption describes a situation in which a test feature fails to be observed. For example, an *Order Management* agent in an electronic bookstore system is specified in the design to only accept purchasing orders with no more than ten books. An assumption can be defined, specifying that if an instance of the *Order Management* agent accepts an order with more than ten books, a fault will be identified.

Estimation of resource and budget, and negotiation of testing schedules are also performed in the planning stage. However these activities are related to management of testing and mainly depend on the application and human views. Hence they will not be discussed in this thesis.

- **acquire the test set**

  In this stage test cases are designed and implemented for unit testing the SUT. The generation of test cases is based on the test features and the fault model specified in the first stage. There are various approaches available for the generation of test cases. They describe the techniques utilised for different aspects of test case generation, such as how the set of test inputs is generated, how to determine the steps performed by a test case and how to determine the pass and failure criteria of a test case based on the knowledge about the SUT. Determination of appropriate testing techniques depends on the characteristics of the SUT, taking into consideration views of developers.

  For example, some techniques generate test cases based on the information retrieved from a structured specification of the SUT. They are known as *specification based* techniques [Juristo et al., 2006]. Such a specification can be of different forms, such as a well organised document that describes functionalities of the system, a model of a finite-state-machine that specifies the state transitions of the system, or a system design model that is developed based on a software development methodology. Typically, some techniques generate input of test cases according to the value domain information

There are also other sorts of techniques for test cases generation, such as code based techniques which study the implemented code of the SUT for generating test cases. Some techniques concentrate on the structure of the source code, generating test cases to cover sentences, conditions or branches in the source code as comprehensively as possible [Hutchins et al., 1994; Roper, 1997; Frankl and Iakounenko, 1998; Frankl and Deng, 2000]. Some other techniques consider the possible coverage of all data related items defined in the code, such as variables and database [Weyuker, 1990; Mathur and Wong, 1993].

During this stage test procedures are also specified, each of which describes how test cases are organised and executed in an appropriate way to test a unit. A test procedure is usually in the form of a sequence of steps performed by testers to execute test cases, where some of the steps may be iterative if necessary. Testers may be either humans or programs that can execute test cases.

An optional activity of test design is the implementation of test cases and their preconditions, which is necessary when the test cases are in the form of programs that can be automatically executed. This activity sometimes is separated as an independent stage termed test implementation [Fewster and Graham, 1999, page 13; Mosley and Posey, 2002, page 15]. In this stage, test cases are developed as programs that can be executed. Furthermore, preconditions for the execution of test cases are also implemented prior to the execution of the test cases. For example, if test cases need to connect to a server program, such a server or its simulation should have been implemented and be functional before the execution of test cases.

Moreover, some auxiliary code is often developed in order to facilitate test case execution. Auxiliary code usually performs the tasks that are more easily performed by a program than by human developers, such as generation and execution of a large number of test cases and automated analysis of test outcomes. More details about developing auxiliary code for the purpose of testing will be discussed in section 2.5.2.
• measure the test unit

This stage is commonly called test execution because in this stage test cases are executed following the test procedures specified and outcomes are analysed to reveal faults. Test cases need to be executed in a regressive way, which means that test cases are rerun after detected faults are fixed in order to check if as a result of changes previously detected faults have been really removed or if new faults appear. Test cases are executed and re-executed until a set of pre-defined end conditions have been reached. For example, the number of detected faults has been reduced to match an acceptable criteria, or there are no serious faults detected. Such end conditions are commonly specified subjectively by developers, according to the quality requirement of the SUT and how urgently the product is required to be deployed.

A unit testing process for a SUT always includes at least partially the activities discussed above. Some of these testing activities can be automated in order to improve the efficiency of the testing process. The next subsection introduces the background on test automation and relevant approaches.

2.5.2 Test Automation

Test automation in general is the use of programs to automatically perform some or all activities of a testing process. For example, the developer can manually develop test cases in the form of a program that is then executed to automatically test a SUT. Alternatively a testing program can be developed to automatically generate test cases based on the design specification of a SUT and also execute these test cases. According to the work of Binder [1999, page 802] and Fewster and Graham [1999, page 9], test automation is beneficial because:

1. It can efficiently verify that detected faults have been fixed.
2. It guarantees the consistency and repeatability of a testing process.
3. It facilitates the testing of cross-platform applications.
4. It is an efficient way to reduce the time it takes for a testing process.

Unit testing via automated means is the recommended approach, as the benefits above are most significant in automated unit testing. Runeson [2006] states that “test automation and tailoring frameworks for unit testing are successful practices.” in his survey of unit testing
methods performed in 19 companies. He concludes that the benefits and characteristics of automation facilitates unit testing processes. A unit under test is often a program. The features related to a unit under test are more easily accessed and controlled by the test code of automated test cases than by human testers, such as the input to the unit, the execution of the unit and the outcomes produced by it. Furthermore, the development process of a system will become more flexible with automated unit testing. For example, following some popular software development paradigms such as Test-Driven Development [Beck, 2003], unit test cases for a unit are developed simultaneously with, or even before the unit is developed. The unit can therefore be tested immediately after it is developed to verify if it works properly. When the unit is modified in the future, developers can also verify efficiently whether the modification introduces new faults or not by running the test cases to test the unit again. By doing this faults are captured as early as possible.

Automation can be applied to different testing phases to varying degrees. The phases of test planning and design of the principles for generation of test cases are intellectual activities by nature [Fewster and Graham, 1999, page 18; Binder, 1999, page 58]. They are difficult to automatically perform via test code in that the quality of these activities strongly depends on the views of the developers and accumulated working experience. The activities in these phases require creativity to be demonstrated by human developers, such as the identification of test features, the determination of testing approaches, the definition of testing procedures and the determination of strategies for selecting appropriate test cases. These intellectual activities are rarely repeated, they are performed only once under most circumstances, unless the software is modified because of changes in requirements. Therefore, test planning and design of test cases are the phases that are normally not automated.

Activities in the phases of test implementation and execution are clerical in nature, such as generation of test cases, execution of test cases and generation of a test report. They are performed after test strategies have been determined. Less creativity or intelligence is required for these activities. In addition, they are usually performed regressively, being repeated many times. Therefore activities in these phases are worth automating.

Automation of test implementation requires appropriate strategies that guide automated generation of test cases and there have been various approaches at present. For example, random approaches that generate test cases based on assumptions about the distribution of possible failures that may exist in a SUT [Avritzer and Weyuker, 1995]; path-oriented approaches generate test cases for covering selected execution paths from the control-flow graph of a SUT [Korel, 1990; Beydeda and Gruhn, 2003]; goal-oriented approaches generate
CHAPTER 2. BACKGROUND

test cases for achieving a selected goal such as executing a statement of the implemented code or reaching a branch of the program’s control flow [Ferguson and Korel, 1996; Pargas et al., 1999]. The strategy of model based testing [Apfelbaum and Doyle, 1997] derives test cases from a model that describes some aspects of the SUT for testing purposes. Our research follows the principles of model based testing for testing agent systems, which will be introduced in the next section.

Automation of test execution needs the development of test cases that can be automatically executed. There have been a series of coding frameworks supporting the implementation of automated test cases by different languages, such as the Google C++ Testing Framework 3 for the C++ language, JUNIT [Tahchiev et al., 2010] for the Java language, NUNIT [Hunt and Thomas, 2007] for Microsoft .NET programming languages 4. Furthermore, automated test execution also requires the implementation of appropriate infrastructure that automatically controls a testing process including execution of test cases and analysis of testing outcomes. Such an infrastructure is usually known as test harness [Binder, 1999, page 48], as introduced in the following.

Test Harness

The automation of a testing process requires the development of some auxiliary code that automates some (or ideally, all) activities of test implementation and execution. As the modules that are invoked by the program under test 5 may be incomplete at the time when the unit is tested, it is also necessary to develop some auxiliary code that simulates those modules in order to ensure the program under test executes properly when invoking such modules. The auxiliary code developed to support the test automation of a SUT is called its test harness [Binder, 1999, page 48; Burnstein, 2002, page 148; Dustin, 2002, page 191], which in general consists of these main components as follows [Binder, 1999, page 962; Burnstein, 2002, page 148]:

- A Test driver,
  which performs functionalities for driving a testing process. It automates the tasks of the testing process including the setting up of a test environment, the generation of test cases, the control of testing procedures and clean-up operations such as restoring the environment to the original state.

3http://code.google.com/p/googletest/
5the SUT or part of the SUT
CHAPTER 2. BACKGROUND

- **Stubs (or Mocks),**
  which simulate the programs that will interact with the programs under test but are not developed yet. Therefore stubs are necessary to simulate those programs. Such programs can be internal modules in the SUT or external programs that interact with the unit under test (e.g. an external server). They are developed in order to verify the capability of the unit under test for interacting with those modules.

- and **Built-in-Test Code (BIT Code or Test Code),**
  which is code embedded into the implementation of the SUT in order to check runtime states and perform some testing specific operations. With the use of BIT code runtime states of the program under test can be tracked and some operations for the purpose of testing can be performed.

The general structure of a test harness is expressed in Figure 2.4.

Test harnesses provide testers with a method to automate or semi-automation a testing process. The degree to which automation is applied to a testing process depends on the functionalities of the testing harness that is implemented and strongly depends on what testers are trying to achieve. For example, the test harness for a unit under test can be developed as either a set of automated test cases that still require testers to manually execute them, or a completely automated testing framework that covers both generation and execu-
CHAPTER 2. BACKGROUND

The development of a test harness is also an important part of the research in this thesis. Chapter 4 will discuss in detail our approach for developing appropriate test harnesses for unit testing an agent system.

2.5.3 Model Based Testing

It has been widely accepted that Model based testing is a strategy for how to derive test cases in whole or in part from a model that describes some (or all) aspects of the SUT, such as the SUT’s behaviour, states and the range of input values [Apfelbaum and Doyle, 1997; Dalal et al., 1999; Rosaria and Robinson, 2000; El-Far and Whittaker, 2001; Reuys et al., 2005]. It is a kind of specification based testing as discussed in Section 2.4.2. A model within the paradigm of software development can be in various forms, such as a design document that is created following the UML (Unified Modeling Language) [Grady et al., 2005] specifying all design details of a system, or a formalised specification written by the Spec# modelling language in Microsoft SpecExplorer which is a development tool for model based testing.

Models can be utilised in many ways throughout the development process of a software system, including quality improvement of specifications, code generation, reliability analysis and test case generation [Apfelbaum and Doyle, 1997]. Model based testing concentrates on the use of models for test case generation. This section will discuss in detail the characteristics of a model for the purpose of testing (or, test model in short) and the research on model based testing.

Test Model

A test model describes some aspects of the SUT and is generally simpler than the system it describes. The motivation for using test models comes from the advantage that test cases can be generated and checked based on a model that is simpler than the system itself and easier to analyse, so the cost of test case generation can be reduced [Apfelbaum and Doyle, 1997; Binder, 1999, page 112]. However, a test model should also contain enough information to help developers to derive test features and produce test cases.

In general, it is suggested that a test model to have the following features [Binder, 1999, page 117; Utting and Legeard, 2006, page 9]:

1. A test model should be smaller and simpler than the system it describes.

CHAPTER 2. BACKGROUND

2. It is detailed enough to accurately describe the features to be tested.

3. It preserves details that are useful for revealing faults.
   
   For example, it is suggested that a test model should contain the information from
   which expected outcomes can be derived.

4. It represents the inputs to and the outputs of the SUT, so that testers (human testers
   or test programs) can, during the testing process, generate the inputs and determine
   whether required outputs are produced by the program under test.

5. It represents state transitions of the SUT, so that testers can determine what state has
   (or has not) been achieved by the system when it is tested.

Some case studies on model based testing have reflected that the use of test models is
an effective way to improve the quality of testing activities. A case study of Apfelbaum and
Doyle [1997] shows that by using a test model for capturing system requirements and for
automated test case generation, the whole testing process was completed in 12% of the time
required without the use of a test model. Robinson [2000] states that a test model helps to
construct the system’s behaviour early in the development cycle and to reveal ambiguities
in the specification and design. Another case study shows that by using a test model, more
test cases can be generated and applied to the SUT and the number of detected faults can
be increased by 11% [Pretschner et al., 2005].

Development of a Test Model

During the development process of a software system, a design model is often created that
specifies the details of the system. For example, the UML model of a system contains a series
of diagrams that describe the architecture and behavioural aspects of a system. However,
the information described in a development model sometimes is not sufficient for test case
generation [Utting and Legeard, 2006, page 31]. For example, the range of the input values
of modules, which are critical for test input generation, are not always specified precisely in
most development models. For another example, the processing information described in the
development model of a system is often insufficient for test case generation.

A design model is often not precise enough for test case generation. So current ap-
proaches on model based testing usually follow two strategies for the development of test
models: development of specific test models or use of existing design models that have been
supplemented with extra testing specific information.

38 (October 14, 2011)
CHAPTER 2. BACKGROUND

The development of a specific test model is concentrated on developing a test-ready model that is specific for test case generation. Such a model can be in various forms, such as a finite state machine, a statechart, a UML model or Markov chains [El-Far and Whittaker, 2001]. A test model can be developed in different ways. For example, Apfelbaum and Doyle have introduced an approach to develop a behaviour model for the SUT, which helps testers to understand and predict the system’s behaviour [Apfelbaum and Doyle, 1997]. Test cases can be derived from the model to verify user scenarios. Another example is the approach of data-model based testing discussed by Dalal et al. [1999]. A test model developed following their approach focuses on data issues, specifying the inputs to and the outputs from the SUT. The algorithm for test case generation focuses on the generation of input value combinations and the relations between inputs and outputs. Further, some researchers use Markov Chains [Kemeny and Snell, 1960] for building a model which describes the state transitions of the SUT and based on which test cases are derived for verifying the SUT’s states [Whittaker and Thomason, 1994; Sayre, 1999; Whittaker, 2000; Walton and Poore, 2000].

A test model can also be developed by using some programming languages that are specific for the purpose of testing, such as the TTCN-3 (Testing and Test Control Notation) language [Grabowski et al., 2003]. TTCN-3 is a popular testing language that originated in the telecommunications area and has been improved for software testing in the third version. TTCN-3 can be used for developing a behaviour model of the SUT, from which test cases can be easily extracted.

The development of specific test models leads to high concentrations of specific models for the purpose of testing. However, it may not be efficient when the existing design model of the SUT contains sufficient information that can be used for testing. This is because extra cost of time and resources will be required. Therefore the use of an existing development model supplemented with additional testing-specific information is another strategy for developing a test model.

There have been popular development models for software development in recent decades, such as the UML (Unified Modeling Language) model [Grady et al., 2005], the UCM (Use Case Map) model [Buhr and Casselman, 1996] and the BDI (Belief Desire Intention) agent model [Bratman, 1987; Rao and Georgeff, 1991; 1992]. Some researchers have focused on making testability extensions based on design artifacts of a development model. For example, Briand and Labiche have developed an approach to generate test cases based on the test requirements that are derived from the formalised UML use cases of a SUT [2001]. Binder has studied most design artifacts of UML diagrams to explore how these artifacts can be
used for test design. He has made testability extensions for the kinds of design artifacts which do not contain sufficient information for test case generation [Binder, 1999, page 269]. Another case of model transformation explores a method to covert the SUT’s UCM model, which describes the system’s behaviour and operational requirements using graphical use case scenarios, to a TTCN-3 behaviour model [Amyot et al., 2005]. These approaches utilise the existing development model of the SUT and avoid the cost of time and resources for developing a new test model.

Derivation of Test Cases from a Test Model

There are various techniques for the derivation of test cases from a test model. Utting and Legeard [2006] have summarised some strategies for deriving test cases based on the information described in a test model. Following these strategies, test cases can be generated based on:

1. The domain description in a test model
   
   This strategy is applied when the test model of a SUT describes the domain information, such as the value ranges of the inputs to the modules under test. The inputs of test cases can be generated by using some algorithms to rationally select appropriate combinations of input values. For example, if a unit to be tested has multiple input variables and the domain information of these variables has been described in the test model, the developer can use some techniques to generate a set of input value combinations that is a manageable scope, such as the techniques of boundary value checking [Copeland, 2004, page 39], equivalence class partitioning [Burnstein, 2002, page 67; Jorgensen, 2002, page 39] or combinatorial design [Cohen et al., 1997; Blass and Gurevich, 2002; Tai and Lie, 2002; Grindal et al., 2006; Kuhn et al., 2008] Each value combination can be used as the input of a test case.

2. The behaviour description in a test model
   
   If a test model describes the SUT’s behaviour in some way, test cases can be derived from the model to verify the correctness of the SUT’s behaviour. For example, a test model sometimes provides sufficient information about the relationship between the inputs to and outputs from a unit under test. According to the relationship specified, test cases can be derived to contain the expected output values of the unit under test given particular input values [Apfelbaum and Doyle, 1997]. By doing this test cases can be executed to verify if the unit under test gives outputs as expected.
CHAPTER 2. BACKGROUND

For another example, an activity diagram in the UML design model of a system represents sequences of actions that describe an activity that may occur. Binder has extended a UML model by specifying a set of test requirements that verify the relations between the actions described in an activity diagram [Binder, 1999, page 296]. For instance, a test requirement states that if one action is specified preceding another action in an activity diagram, the former should happen being followed by the latter at runtime. Test cases can be generated according to this requirement for verifying the relevant system activity.

3. The description of state information in a model

Some researchers have developed state-based test models for a SUT describing the SUT’s state transition, such as a model of Finite State Machines (FSM) [Utting and Legeard, 2006, page 69], a usage model that describes what operations the user is likely to perform to the SUT and the relevant state transitions [Sayre, 1999; Prowell, 2003], a test model of formalised use cases following the UML standard [Briand and Labiche, 2001] and a model of UML state machines [Grady et al., 2005]. They have explicitly explored approaches for deriving test cases from those test models in order to verify the state transitions of the SUT.

The use of an appropriate strategy for deriving test cases depends on what information is described in the test model of the SUT. The above strategies can also be used together when a test model contains various information about the SUT as discussed above. In chapter 5 an approach will be introduced that derives test cases based on various information described in the test model of an agent system.

2.5.4 Testing Agent Systems

There have been approaches developed in recent decades for testing agent based systems. These approaches can be discussed from three aspects. Firstly appropriate test models have been widely used for test case generation in these approaches [Rouff, 2002; Knublauch, 2002; Caire et al., 2004; Seo et al., 2004; Zheng and Alagar, 2005; Tiryaki et al., 2007; Nguyen et al., 2008a], though these test models are developed in different ways. In these test models, the agent specific characteristics are described in detail so that test cases can be generated for verifying these characteristics. Secondly most approaches for testing agent systems implement or partially implement some automation for a testing process, but none
of them have implemented a complete testing framework including both automated test case generation and execution. Lastly these approaches test agent systems in different scopes. Some of them verify the behaviour and states of a single agent, taking into consideration communication between agents [Knublauch, 2002; Rouff, 2002; Caire et al., 2004; Zheng and Alagar, 2005; Nguyen et al., 2008a; Coelho et al., 2006]. While some others explore the internal architecture of an agent, testing its internal components such as plans [Tiryaki et al., 2007].

As discussed in section 2.5.3, there are mainly two different ways for the development of a test model: the use of an existing design model that has been supplemented with extra testing specific information, or the development of a specific test model. Test models for agent systems usually are also developed in these two ways. There are also some approaches that do not use particular test models in their testing processes. In this section current approaches for testing agent systems are compared based on these three ways, taking into consideration the degree to which test automation is applied to a testing process and the scope to which an agent system is tested.

Use of Existing Design Models

Various agent system development methodologies such as Tropos [Bresciani et al., 2004] Gaia [Wooldridge et al., 2000; Zambonelli et al., 2003] and others, have well developed structured models that could be used for test case generation. Several agent platforms do already use design models for some level of assistance in generation of test cases.

For example, eCAT is a testing tool that applies a goal-oriented testing approach for testing an agent system [Nguyen et al., 2008a,b,c; 2010] 7. The approach derives test cases from the goal model of the SUT developed using the Tropos methodology. Such a goal model specifies the goal hierarchy of the SUT, describing how goals are decomposed to elementary goals. Positive test cases and negative test cases are then derived from the goal model to respectively verify the fulfillment (or not) of every elementary goal. The process of test case derivation is partially automated, as the skeleton code of test cases are provided by the tool and then completed by the developer. The execution of test cases is automated by using a Tester Agent.

eCAT includes an input generator that generates the inputs of test cases based on the interaction ontologies of agents [Nguyen et al., 2008c]. Interaction ontologies define content

7eCAT has been developed simultaneously with the work presented in this thesis.

42 (October 14, 2011)
semantics of agent interactions, which describe concepts understandable by agents that interact. For example, the BookBuyer agent could send a Buy a Book message to the BookSeller agent. The interaction ontology of the BookBuyer and BookSeller agents may include the information that the message should contain the properties of title, author and price and a restriction rule that the price property should be within 0 and 100. The input generator in eCAT, following a series of generation rules, can create valid and invalid test inputs based on agent interaction ontologies. For the example above, the input generator can, based on the ontology information related to the Buy a Book message, generate a valid input message with the price 50 and an invalid input message with the price 101.

For another example, Knublauch’s approach [2002] tests an agent system based on the design model of the Gaia methodology [Zambonelli et al., 2003]. He introduces a set of APIs that are extended from the JUNIT testing framework [Tahchiev et al., 2010], taking into account the details of a Gaia design model. The developer can use these APIs to develop the test cases that are automatically executed to verify the capability of an agent to interact with others via messages. However, test cases are still required to be manually generated. Therefore a testing process following this approach is semi-automated. Also, this approach does not explore agents’ internal units.

Caire et al. [2004] extend the design model of an agent system developed using the PASSI methodology [Burrafato and Cossentino, 2002] for the purpose of test case derivation. The researchers introduce MAZBD (Multi-Agent Zoomable Behaviour Description) diagrams that describe behaviours of agents more precisely than the design specification. Their testing framework also provides a set of APIs and skeleton code based on which the developer can develop test cases for verifying agent behaviours, as well as a tester agent that can automatically execute test cases. The testing process is also semi-automated as test cases are required to be manually developed.

The SEAGENT [Dikenelli et al., 2005] model of an agent system can also be used for testing an agent system. Tiryaki has developed the SUNIT [Tiryaki et al., 2007] testing framework for testing the agent systems developed by the Seagent platform [Dikenelli et al., 2005]. SUNIT, which is extended from the JUNIT framework [Tahchiev et al., 2010], specifies a set of APIs for human developers to manually develop test cases that verify the hierarchy of the plan-level structure and actions performed by plans.

There are also some other design models that can be used for testing agent systems. Although they are not agent specific. Seo et al. [2004] have explored an approach for deriving test cases from an extended Statecharts model of an agent system. Statecharts is origi-
nally a visual and behavioural specification language for modeling reactive systems [Harel, 1987; Harel and Naamad, 1996]. The researchers first extend Statecharts to allow flexible description of autonomous behaviours of agents. They then discuss the details on how to automatically generate test cases based on such an extended Statecharts model of an agent system. However, there is no further discussion on the automation of test case execution in their approach.

**Development of Specific Test Models**

Some other testing approaches are based on explicitly developed test models for the generation of test cases. For instance, Rouff [2002] has introduced an approach to develop a message based model of an agent system. From such a model test cases can be derived to verify communications between agents. In his testing framework, a *test agent* is developed to perform an automated testing process. The test agent automatically extracts from the message based model of a SUT the interactions of every agent with others, and then tests every agent to verify if it replies with correct messages when receiving a particular one. Zheng and Alagar [2005] develop an ESM (Extended State Machine) model of an agent system to formalise the behaviours of agents. They then introduce an approach to automatically generate test cases based on the ESM model of an agent system.

**Without a Test Model**

There are also some approaches that do not use particular test models for the generation and execution of test cases. For instance, Coelho et.al develop the JAT testing framework [2006] that is based on a set of fault types relevant to general agent features. The framework provides skeleton code for human developers to manually develop test cases for testing an agent system developed by the JADE platform [Pokahr et al., 2003]. A *tester agent* is also used in the framework to automate the execution of test cases. In addition, *mock agents* are used to play the roles of other agents that are invoked by the agent under test.

**Summary**

In conclusion, current testing approaches for agent systems review different features of agent systems, build test models in different ways and apply test automation to different degrees. None of these approaches applies complete automation for a testing process from test case generation, test case execution until analysis of test outcomes and output of testing results,
and only one of them explores internal characteristics of units within an agent. The following chapters of this thesis introduce an automated unit testing framework that has been developed, which explores internal units within an agent and realises complete automation for the testing process of an agent system.
Chapter 3

Test Units

Our research concentrates on unit testing the components of a single agent. This chapter discusses the natural units of an agent based system for unit testing. Unlike more traditional software systems, such as those based on Object-Oriented principles, where the base units are classes that are called via method invocation, the units in agent systems are more complex in the way they are called and executed. For instance, plans are triggered by events and an event may be handled by more than one plan. Plans may generate events that trigger other plans either in sequence or in parallel. A testing framework for agent based systems must take these details into consideration in identifying the appropriate units and developing appropriate test cases. In this chapter, test units within an agent are identified by exploring the structural hierarchy of a single agent based on the Prometheus methodology. Then the fault model is defined, which describes what faults are likely to be found, for each type of unit according to its agent-specific characteristics. Based on test units and the fault model identified, test cases will be designed and testing procedures will be specified. Chapter 4 and chapter 5 will describe the details on testing procedures and test cases.

3.1 Units in Agent Systems

Figure 3.1 outlines the components of an agent that is developed using the Prometheus methodology. These components are typical for most systems that follow the Belief Desire Intention (BDI) model of agents, which has been discussed in Section 2.1. According to such a structure, an agent naturally contains plans, events and beliefs. These components may also be packaged into sub-units called capabilities. However as capabilities are simply collections of plans, events and beliefs, we do not further consider testing of capabilities.
CHAPTER 3. TEST UNITS

Figure 3.1: Agent Components in Prometheus

A **plan** is triggered by an event taking its context condition into consideration. It executes a series of steps for achieving its goals and may succeed or fail depending on the state of the environment. A plan may also post some events for activating its subtasks during its execution. Verification of such features gives the developer the confidence that the plan works properly. Hence **plans** are identified as a type of basic unit for the testing purpose.

An **event** is the generalisation of a percept, a message or an internal event defined in the Prometheus methodology. In an agent system developed using Prometheus, **percepts** are external inputs to the system, **messages** are interactions between agents and **internal events** activate subtasks within an agent. They are always generalised by the concept **events** as discussed in section 2.3. An event triggers one or more plans for achieving a particular goal. Its ability for triggering appropriate plans needs to be verified. Hence **events** are also identified as a type of test unit.

The other low level unit that is important in the agents’ processing is a **belief**. Changes of a belief, such as the insertion of new information or the updating of existing information, may automatically trigger events to activate other plans. These actions and their associated changes in the belief can be specified in the design descriptor of the belief as action rules (refer to section 2.3) and must be implemented correctly if specified. In addition, the structure of a belief, which consists of a set of data fields, must also be verified to guarantee that it is implemented as specified in the design documentation. Hence **beliefs** are considered as a type of testable unit.

Given the above discussion, the basic units identified within an agent are thus **plans**, **events** and **beliefs**.
3.2 Fault Model

An approach of fault-directed testing is explored in this thesis, where we intend to identify faults in the implementation of the SUT through failures. This is in contrast to conformance-based testing, which tests whether a SUT meets the business requirements. When a unit is tested, we aim to reconcile inconsistencies between the actual behaviour and the expected/designed behaviour of the unit under test, because such inconsistencies indicate failures in the implementation or in the design documentation. As discussed in section 2.3, the Prometheus design descriptor of a unit specifies expected features of the unit. If the unit under test does not perform actions or achieve states as specified in the design, or if it performs actions or achieves states that are not specified, faults can be identified. For example, we may identify a fault if a plan under test posts an outgoing event that is not specified in the design, or the plan never posts an outgoing event that has been specified in the design. Such faults indicate problems existing in the SUT.

In addition, there may be some situations that do not directly indicate inconsistencies between implementation and design, but may still be possible sources of errors. They are also considered as a kind of fault in our testing framework because it is considered useful to identify them for checking, though they may not be actual errors in the design or in the implementation. For example, a common cause of error is incorrectly specifying context conditions of the plans that handle an event [Poutakidis, 2008, page 57, 59]. This can result in more than one plan matching the same situation and wrong plans being selected. The situation in which more than one plan is applicable is called overlap in Prometheus. Although the designer can specify in the design specification that overlap is expected for a particular event, it is still difficult to know in what situation overlap is expected. Therefore, initially we identify all overlap situations as warnings, so the designer/tester can investigate further.

Sometimes a fault may be incorrectly reported due to inadequate coverage of the test cases generated. For example if an expected outgoing event has not been posted, it may not be an actual error if the conditions under which the message is posted are not covered by the test cases generated. In our framework the test cases for a particular unit are automatically generated, or manually added by the human tester. Although we have used an approach to generate test cases as comprehensively as possible (refer to chapter 5 for details), some necessary cases may still be missed. Where this may be the cause of an apparent error, a warning will be generated. If it is not an error the tester can add an additional test case as appropriate.
In order to realise fault-directed testing, we require knowledge about the failures that can be detected. This knowledge is often called the fault model as discussed in section 2.4. We first need to explore features of each type of unit that needs to be verified, then study all possible points of failure associated with these features. We term such features testable features and such possible points of failure fault types. We also categorise faults into different levels in order to distinguish if they are definite errors or just possible sources of errors. Based on this fault model test cases can be generated to perform the fault-directed testing by looking for these failure points.

Testable features for each unit type are identified by studying the internal structure and behaviour of the unit type described in the associated design artifacts. For example, a plan usually contains a triggering event, a context condition and a plan body. We study these components to identify the features that should be verified. In addition, the behaviour of each unit type is taken into consideration when identifying testable features, such as events being posted from a plan or triggered by belief changes. Other characteristics of a unit type are also studied to guarantee each unit type is considered as comprehensively as possible for the identification of testable features.

We also define a set of fault types for each unit type according to its testable features. A fault type describes possible points of failure in the SUT associated with a particular type of unit. For instance, a fault type described as “The plan is not triggered by its triggering event as specified.” might be associated with the testable feature “Does the plan get triggered by the event that it is supposed to handle?”. A fault of this type demonstrates a situation in which the relevant testable feature is not correctly implemented. In our testing framework, a set of test cases are generated to test a particular unit, checking the various testable features of the unit.

Fault Levels

As discussed some faults indicate definite errors of the SUT, while others indicate only possible errors. We categorise faults into different levels to distinguish this difference. This categorisation is necessary because faults of different levels may affect the continuation of the testing process during which all units within an agent are tested. If a fault that is identified when a unit is tested is a definite error, other units that depend on the unit should no longer be tested. On the other hand, if a fault is not a definite error, two options will be possible: other units that depend on the unit are still tested continuously; or the testing process is
CHAPTER 3. TEST UNITS

terminated and the developer investigates the fault. Therefore a fault should be assigned to
a level so that appropriate operations can be applied when the fault is detected.

We define three levels for faults below:

- **Level-3 Fault** (Warning):
  This is often an error, although in some situations it may not be. When a level-3 fault
  is identified, the developer is given a warning message which describes the fault, so it
  is termed a *warning*. The developer/tester can indicate if it is not error, in which case
  it can be tagged so no further warnings are generated for that case.

- **Level-2 Fault** (Error):
  A level-2 fault can be directly asserted as an error. For example, if the context condition
  of a plan is absent in the design, the plan should be always applicable when its triggering
  event is received. Hence an error will be identified if the plan is not applicable in some
  test case. An error indicates inconsistencies between design and implementation, so
  changes to correct the error can be in code or in the design.

- **Level-1 Fault** (Exception):
  Faults of level-1 are exceptions, which are thrown up by the SUT at runtime. *Exceptions*
  are not agent specific, but the occurrence of an exception must be caught in order to
  avoid the runtime execution of the SUT being terminated. We have designed our test
  harness to catch any exceptions that are thrown up when a unit is tested and categorise
  them as level-1 faults. Hence a level-1 fault is also termed an *exception*.

Categorisation of some faults (but not all) of level-2 assumes sufficient test cases. How-
ever, we can never be certain that relevant input data is not missed. Hence such faults should
be classified as level-3 as the test cases generated never have full coverage. The developer
may need to manually add suitable test cases to eliminate the faults. We will show such
changes in fault levels when introducing particular fault types in the following sections.

Based on the general discussion above on the fault model, testable features are identified
for each type of unit as well as the associated fault types.

3.2.1 Plans

A plan in general consists of a *triggering event*, a *context condition* and a *plan body*. The
*triggering event* of a plan indicates the plan’s relevance to the event. The *context condition*
of a plan determines the applicability of the plan with respect to the agent’s beliefs about
the current state of the world. The plan body outlines a sequence of steps for achieving a
particular goal. These steps may be subtasks, which are activated by posting internal events
or external message events. The former are handled by the agent that contains the plan
while the latter are handled by another agent. These aspects of a plan unit are analysed and
four testable features are derived. Their associated fault types are outlined in Table 3.1.

<table>
<thead>
<tr>
<th>Associate Feature</th>
<th>Fault Type</th>
<th>Fault Description</th>
<th>Default Level</th>
<th>Change of Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>1.1.1 FT_TRIGGER</td>
<td>The plan is not considered based on its triggering event as specified.</td>
<td>2</td>
<td>→ 3</td>
</tr>
<tr>
<td>1.2</td>
<td>1.2.1 FT_CC_ABSENCE</td>
<td>The plan is not applicable in some test cases even though the context condition (CC) is absent in the design.</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2.2 FT_CC_VALID</td>
<td>The CC value is always evaluated as true when the CC has been specified in the design.</td>
<td>2</td>
<td>→ 3</td>
</tr>
<tr>
<td></td>
<td>1.2.3 FT_CC_INVALID</td>
<td>The CC value is always evaluated as false when the CC has been specified in the design.</td>
<td>2</td>
<td>→ 3</td>
</tr>
<tr>
<td>1.3</td>
<td>1.3.1 FT_OUTEV_NEVER</td>
<td>A specified outgoing event is never posted out.</td>
<td>2</td>
<td>→ 3</td>
</tr>
<tr>
<td></td>
<td>1.3.2 FT_OUTEV_NOT</td>
<td>An event posted at runtime by the plan is not specified as an outgoing event in the design.</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>1.4.1 FT_COMPLETION</td>
<td>The plan fails to complete in some test cases.</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

* The fault level can change if the test cases generated do not have full coverage.

Table 3.1: Fault Types of Plans

**Feature 1.1:** *Is the plan considered based on the event that it is supposed to handle?*

The plan should be considered for applicability as long as the plan is relevant to handle its
triggering event as specified in the design. This requires the event to be of the correct type
and also that any required aspects of the event attributes hold. If not, a fault of the plan
not considered based on its triggering event will be identified (refer to FT\_TRIGGER in
Table 3.1). However, it is possible that the set of test cases does not include the necessary
criteria for the required attributes of the event and consequently the plan is not triggered by
the event. Hence the fault type becomes level-3 without the assumption of the full coverage
of test cases.
CHAPTER 3. TEST UNITS

**Feature 1.2:** Does the context condition of the plan provide discrimination if present?

The context condition of a plan indicates under what conditions the plan is applicable for the event it handles. The absence of a context condition denotes that the plan is always applicable in all situations. If the developer specifies a context condition for the plan, then the context condition should be evaluated as valid in some situations and as invalid in others.

There are therefore three possible failure points associated with this feature as outlined in Table 3.1. These are all categorised as level-2 if full coverage of test cases is assumed, but two of them are changed to level-3 as the test cases generated may not have full coverage.

**Feature 1.3:** Does the plan post the outgoing events specified and only those events?

There are two possible failure points for this feature: an outgoing event specified in the design is never posted during any test case; and the plan under test posts a message that is not specified in the design in some test case. These two failure points are respectively associated with two fault types as shown in Table 3.1: a specified outgoing event not posted (FT_OUTEV_NEVER) and a non-specified event being posted at runtime (FT_OUTEV_NOT).

A fault of non-posting of a specified outgoing event is changed to level-3 without the assumption of full coverage. This is because the posting of an outgoing event may be conditional and the test cases generated may not include the necessary condition, rather than an actual error of missing or unreachable code for posting the message.

A fault of a non-specified event being posted at runtime is categorised as level-3 in that the developer may implement extra outgoing events for development purposes, such as a debugging message that reports on the runtime states of a plan.

**Feature 1.4:** Does the plan complete?

In the normal program execution, there may be some reasons that lead to the failure of the plan under test, such as changes in the environment after the time the plan is selected. However, in the controlled testing environment all plans that have been selected for execution should complete. Hence if the plan under test fails to complete, there is assumed to be an error in its implementation and is detected as a fault of level-2 (refer to FT_COMPLETION in Table 3.1).

52 (October 14, 2011)
3.2.2 Plan Cycles

In the design specification, the hierarchy of plans is always presented as a digraph, in which a plan points to all its sub-plans via the events posted out by the plan. The interactions of plans with their sub-plans may form a plan cycle within an agent, if a plan posts an event to trigger one of its ancestor plans, such as the example in Figure 3.2. A plan cycle is treated as a special type of plan unit and is tested as a single entity for cyclic-specific features, as follows:

![Figure 3.2: Example of A Plan Cycle](image)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Fault Type</th>
<th>Fault Description</th>
<th>Default Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>2.1.1 FT_CY_EXISTENCE</td>
<td>The cyclic execution does not exists at runtime.</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>2.2.1 FT_CY_NO_STOP</td>
<td>The iterations of the cyclic execution path exceeds the pre-defined maximum limit in some test case.</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.2: Fault Types of Cyclic Plans

Feature 2.1: Does a cyclic execution exist at runtime?

A plan cycle specified in the design implies that a cyclic execution may occur at runtime. For the example in Figure 3.2, $P_1$, which is a descendant of $P_0$ according to the plan hierarchy, may post the event $e_0$ to trigger another instance of $P_0$. This is a case of the cyclic execution. A fault of absence of cyclic executions is identified if the cyclic execution never exists at runtime (refer to FT_CY_EXISTENCE in Table 3.2).

This fault is categorised as level-3, as it is also possible that cyclic execution at runtime is not expected by the developer even when a plan cycle has been specified in the design. For instance, Figure 3.3 presents a plan cycle in a plan hierarchy, which is “$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_0$ ...”. However, the developer may implement these plans in such a way that there are only two execution paths existing at runtime: “$P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3$ ...” (the solid-line
Figure 3.3: A Plan Cycle without Cyclic Execution

(path) and “...P4 → P2 → P0 → P5 ...” (the broken-line path). Such an implementation is possible if the developer implements appropriate logic in the bodies of P0 and P2. For example, P0 activates P5 only when its triggering event e0 is posted by P2. Otherwise P0 activates P1. Similarly, P2 activates P0 only when its triggering event e2 is posted by P4. Otherwise P0 activates P3. Consequently, the cyclic execution path “...P0 → P1 → P2 → P1 ...” never occurs. Hence the absence of a cyclic execution may not always be an error.

**Feature 2.2: Does the cyclic execution path terminate?**

A cyclic execution may continue to infinity without termination. Although the developer may implement some conditions to control a cyclic execution at runtime, it is possible that a cycle is executed without stopping because of some implementation errors. To check this feature, we introduce a pre-defined maximum limit for the number of iterations that occur in the cyclic execution path. The limit value can be specified by the developer in our testing framework. If the cyclic execution exceeds that limit, a fault of exceeding of cyclic execution iterations will be identified (refer to FT_CY_NO_STOP in Table 3.2).

A fault of exceeding of cyclic execution iterations is categorised as level-3 for two reasons. The first reason is that the occurrence of such a fault may not reveal an infinite execution path. The pre-defined limit may happen to not exceed the maximum number of iterations. The second reason is that an infinite cyclic execution may be reasonable within the application. For example, an agent that monitors network status may have a plan cycle that executes infinitely as long as the agent is running.
3.2.3 Events

The testable features of an event have to do with whether there is always an applicable plan to respond to the event (complete coverage) and whether there is more than one applicable plan (overlap). The event under test has incomplete coverage if there is no applicable plan for the event in some situation, as discussed in section 2.3. Overlap and incomplete coverage are common errors in agent systems because the context conditions of the plans that handle the event may not be correctly implemented [Poutakidis, 2008, page 57, 59]. The following is an example of implementation errors leading to erroneous overlap and incomplete coverage:

- A Buy a transport ticket event is handled by two plans: Buy an air ticket Plan and Buy a train ticket Plan, whose context conditions are respectively specified in the design as “The budget is not less than 500 dollars.” (budget ≥ 500) and “The budget is less than 500 dollars.” (budget < 500). If the context condition of the Buy an air ticket plan is incorrectly implemented as “budget > 500”, incomplete coverage will occur when budget is 500 as no plan is applicable; if the context condition of the Buy a train ticket plan is incorrectly implemented as “budget ≤ 500”, overlap will occur when budget is 500 as both plans are applicable.

On the other hand, overlap or incomplete coverage of an event may sometimes be allowed by the application. The developer can specify in the design whether overlap or incomplete coverage is allowed at runtime for a particular event in order to describe the situations in which it is expected to occur. However, such situations are usually described in natural language and are not accessible to the test cases as part of the structured design model. Therefore when overlap or incomplete coverage occurs at runtime we initially flag the situation as a warning and then identify different levels of faults according to the specification in the design.

Hence we verify two features as follows when testing an event:

**Feature 3.1: Is there always an applicable plan for the event?**

If not, the specification in the design will be checked. If the developer has specified that incomplete coverage is allowed for the event, a level-3 fault will be identified, else, a level-2 fault will be identified (Table 3.3).
### Chapter 3. Test Units

<table>
<thead>
<tr>
<th>Associate Feature</th>
<th>Fault Type</th>
<th>Fault Description</th>
<th>Default Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>3.1.1 FT_EV_INCOMP_W</td>
<td>There is no plan applicable for the event under test in some test case and the developer specifies <em>incomplete coverage</em> is allowed for the event.</td>
<td>3</td>
</tr>
<tr>
<td>3.1.2</td>
<td>FT_EV_INCOMP_E</td>
<td>There is no plan applicable for the event under test in some test case and the developer does not specify <em>incomplete coverage</em> for the event.</td>
<td>2</td>
</tr>
<tr>
<td>3.2</td>
<td>3.2.1 FT_EV_OVERLAP_W</td>
<td>There are multiple plans applicable for the event under test in some test case and the developer specifies <em>overlap</em> is allowed for the event.</td>
<td>3</td>
</tr>
<tr>
<td>3.2.2</td>
<td>FT_EV_OVERLAP_E</td>
<td>There are multiple plans applicable for the event under test in some test case and the developer does not specify <em>overlap</em> is allowed for the event.</td>
<td>2</td>
</tr>
<tr>
<td>3.3</td>
<td>3.3.1 FT_EV_NOT_EXEC</td>
<td>A plan that handles the event under test is never executed.</td>
<td>3</td>
</tr>
</tbody>
</table>

**Table 3.3: Fault Types of Events**

**Feature 3.2:** *Is there more than one plan applicable for the event?*

This feature is only relevant to an event that is handled by multiple plans. If the developer has specified that *overlap* is allowed for the event, a level-3 fault will be identified, else, a level-2 fault will be identified (Table 3.3).

If there are multiple plans that are applicable for the event under test, one of these plans should be selected for execution, according to the BDI principle discussed in section 2.1. In general, agent development platforms provide mechanisms for the implementation of such a plan selection, such as the *meta-reasoning* mechanism in JACK. Ideally, the developer should implement the SUT in such a way that each plan that handles the event under test is selected for execution in at least one situation, because it is not sensible to implement a plan that never executes. However, the SUT may be incorrectly implemented leading to a failure that a plan is never executed. For example, in JACK, the user can implement a meta-reasoning plan, which is triggered once the event under test has multiple applicable plans, to select a plan from the applicable plan set for execution. The developer may incorrectly implement the logic in a meta-reasoning plan so that a plan that handles the event under test...
test is never selected for execution.

However, it may be the case that a plan is implemented as an alternative to be used only when another plan fails. If the applicable plan that is selected for execution fails, another applicable plan will be tried. Such an alternative plan may never be executed at runtime if the preferred plan always succeeds. This situation is not an error.

The testing framework needs to detect such a possible failure point of non-execution of a plan that handles the event and the associated testable feature is:

**Feature 3.3**: Does a plan that handles the event under test never executes?

The associated fault type is categorised as level-3 (Table 3.3). Although it is most likely an error, it may have been implemented as a backup plan, or the test cases generated may not cover the situation in which the plan is executed.

### 3.2.4 Beliefs

```java
public beliefset BookDB extends OpenWorld {
    // the beliefset structure
    #key field String bookName;
    #value field float price;
    #value field String author;
    #value field String publisher;
    // A default method “add(String bookName, float price, String author, String publisher)”
    // can be invoked to add a new book record.

    // invoked for querying records in the beliefset
    #indexed query get(String bookName, logical float price,
                        logical String author, logical String publisher);
    ....
}
```

*Figure 3.4: Structure of A Beliefset in JACK (Example)*

A belief is tested for verifying two aspects:

- The first is that the structure of the belief for data storage should be implemented as specified. This verification is implementation specific, as a belief can be implemented in various structures according to different implementation platforms and there are three common kinds. In JACK [Winikoff, 2005] a belief is implemented as a beliefset object with a table structure that contains records, as the example shown in Figure 3.4. In JADEX [Pokahr et al., 2003] a belief is implemented as a collection of one or more
facts, each fact is a particular value of a primitive data type or a class type. A belief is implemented in JADEX by declaring the belief structure and facts in XML format, as the example shown in Figure 3.5. In JASON [Bordini et al., 2007] and 3APL [Hindriks et al., 2001] a belief is implemented as a series of clauses, as the example shown in Figure 3.6. In our research JACK is used as the implementation platform, so we focus on the table-style structure of a belief. In Prometheus a table-style belief is represented as a data entity and each data field in the data entity is a tuple <name, type, key>, such as the example shown in Figure 3.7. Hence we verify that elements (name, type and key) of each data field of the belief should be correctly implemented as specified in the design.

- The second is that a belief may perform an action (e.g. post an event) due to a particular change of the belief. The pair of the particular change and the event activated by the change is specified as an action rule in the design descriptor of the belief, as mentioned in section 3.1. Each action rule is a tuple <operation, event>. Operation denotes a particular operation that changes the belief and is one of insert, delete, update and modify, where modify is a generalisation of insert, delete and update. Event is the event posted by the belief when that particular operation succeeds. Each action rule of the belief should be correctly implemented. Thus we verify the implementation of each action rule for a belief. In general, the correct implementation of an action rule
depends on two aspects: appropriate structure (usually methods) should have been implemented for the action rule and the belief should behave at runtime following the action rule.

To implement an action rule, the change of a belief due to the operation specified in the rule should activate the invocation of particular code (usually particular methods) that then posts the event specified in the rule. This is implemented in different ways according to implementation platforms. For example, in Jadex, Java-based listener methods can be implemented for observing changes of a belief and performing particular actions. In JACK, this is done by the implementation of appropriate callback methods. A callback method is automatically invoked when a particular change occurs in a belief. For example, in JACK the newfact() member method of a belief will be automatically invoked after a record is successfully inserted into the belief. Therefore, to verify the implementation of an action rule, we check that appropriate code (usually methods) has been implemented to be activated by a particular belief change. Our research uses JACK, so we verify if appropriate callback methods have been implemented for each action rule.

Implementation of appropriate code that can be activated is necessary to the implementation of an action rule, but is not sufficient. The developer may have implemented appropriate methods but may not correctly implement the logic of event posting. Hence,
CHAPTER 3. TEST UNITS

### Table 3.4: Fault Types of Beliefs

<table>
<thead>
<tr>
<th>Associate Feature</th>
<th>Fault Type</th>
<th>Fault Description</th>
<th>Default Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>4.1.1 FT, BE, NO, FIELD</td>
<td>A data field is not implemented.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4.1.2 FT, BE, NOSPEC, FIELD</td>
<td>A data field not specified in the design exists in the implementation.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4.1.3 FT, BE, WRONG, FIELD</td>
<td>The Key or Type of a data field is not implemented as specified.</td>
<td>2</td>
</tr>
<tr>
<td>4.2</td>
<td>4.2.1 FT, BE, NO, CB</td>
<td>For an action rule, there is not appropriate callback method implemented.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4.2.2 FT, BE, INAPPRO, CB</td>
<td>For an action rule, callback via the implemented attempt-triggered method does not guarantee the success of the operation specified in the rule.</td>
<td>3</td>
</tr>
<tr>
<td>4.3</td>
<td>4.3.1 FT, BE, NOT, POST</td>
<td>The belief does not post out the event as specified after the operation is successfully applied.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4.3.2 FT, BE, WRONG, POST</td>
<td>The belief posts out the event as specified after the operation fails to be applied.</td>
<td>2</td>
</tr>
</tbody>
</table>

for an action rule we also verify that at runtime the belief should post the event specified in the rule after the operation specified has been successfully applied to the belief.

Therefore, we verify the testable features relevant to the above aspects as follows:

**Feature 4.1:** *Is the structure of the belief implemented as specified?*

The developer may make two kinds of errors when implementing the structure of a belief: neglect to implement some elements of a belief, or incorrectly implement some elements of a belief. In an example of the former, the developer may neglect to implement a data field. In an example of the latter, a data field may be incorrectly implemented with a wrong *Type* or *Key* Hence two fault types are specified respectively associated with these two kinds of errors, as specified in Table 3.4.

**Feature 4.2:** *Is appropriate code implemented for each action rule?*

Each action rule of the belief under test should be correctly implemented. As mentioned above, appropriate code should be implemented so that when the belief under test changes as specified in the action rule, particular code could be invoked to post the event specified in the rule. Hence the source code of the belief under test should be checked to ascertain appropriate code has been implemented for each action rule.

\[\text{Name} \text{ is considered as the identifier. If there is no field in the implementation called the name of a data field specified in the design, we conclude that the relevant data field is not implemented.}\]
CHAPTER 3. TEST UNITS

### Table 3.5: Success-Triggered and Attempt-Triggered Callback Methods in JACK

<table>
<thead>
<tr>
<th>Operation</th>
<th>Success-Triggered Methods</th>
<th>Attempt-Triggered Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>newfact(), modfact(), and moddb()</td>
<td>addfact()</td>
</tr>
<tr>
<td>Update</td>
<td>newfact(), modfact(), and moddb()</td>
<td>addfact()</td>
</tr>
<tr>
<td>Delete</td>
<td>endfact() and moddb()</td>
<td>delfact()</td>
</tr>
</tbody>
</table>

In JACK, we verify whether appropriate callback methods have been implemented for each action rule. A belief operation may activate its relevant callback methods for execution in two ways. Some callback methods are activated only after the associated operation succeeds, while others are activated after the associated operation is applied, no matter if the operation succeeds or fails. We term the two kinds as **success-triggered** methods and **attempt-triggered** methods respectively in order to facilitate the following discussion. The **success-triggered** and **attempt-triggered** methods for concrete belief operations (insert, update and delete) are described in Table 3.5. modify is not a concrete operation and denotes the occurrence of any of insert, update and delete. The situations involving modify will be discussed separately.

For the action rule to be verified, it is expected that the logic of event posting is implemented in a success-triggered method of the operation specified in the rule. By doing this only the success of the operation activates the callback method and consequently leads to the posting of the event. However, the developer may also implement event posting in an attempt-triggered method with additional code to confirm the success of the operation. Hence we check the implementation of callback methods for each action rule under three different circumstances below:

1. **No callback method is implemented.**
   
   Under this circumstance, the operation specified in the rule cannot activate any callback method and the event specified can never be posted. This is an error. Hence a fault of absence of callback methods implemented is identified that is level-2 (refer to FT_BE_NO_CB in Table 3.4).

   If the operation specified is modify, a fault of absence of callback methods implemented should be identified if one of insert, update and delete cannot activate a callback method. Therefore such a fault is identified when there is no relevant callback method implemented for one of insert, update and delete. For example, if the developer only implements the newfact() method, a level-2 fault is identified as the delete operation will not activate any callback method.
2. At least one success-triggered method is implemented and no attempt-triggered method is implemented.

The invocation of the success-triggered method can guarantee the success of the associated operation. Hence we do not identify any fault.

For the modify operation, no fault is identified if there is at least one success-triggered method for each of insert, update and delete, and no attempt-triggered method is implemented. Consequently, each of insert, update and delete triggers a success-triggered method only when it succeeds.

3. At least one attempt-triggered method is implemented

The invocation of an attempt-triggered method cannot guarantee the success of the operation. However, this circumstance may not indicate an error as the developer may add additional code in the method to check the success of the operation, or the developer may have implemented another success-triggered method in which the event specified in the rule is posted. Hence, we identify a level-3 fault of implementation of attempt-triggered methods (refer to FT_BE_INAPPRO_CB in Table 3.4).

For the modify operation, such a fault is identified if for one of insert, update and delete there is one attempt-triggered method implemented. Consequently, the invocation of this particular attempt-triggered method cannot guarantee the success of the associated concrete operation (insert, update or delete). For example, in all situations below a fault of implementation of attempt-triggered methods is identified for the modify operation:

- The developer only implements defact(), endfact() and newfact()
  A delete operation activates defact(), which is an attempt-triggered method and cannot guarantee the success of the operation. Insert or update activates a success-triggered method (newfact()) when and only when it succeeds.

- The developer only implements addfact() and endfact()
  Either insert or update activates addfact(), which is an attempt-triggered method and cannot guarantee the success of the associated operation. Delete activates a success-triggered method (endfact()) when and only when it succeeds.

- The developer only implements addfact() and defact()
  Each of insert, update and delete activates an attempt-triggered method, which cannot guarantee the success of the associated operation (defact() for delete and addfact() for insert and update).
CHAPTER 3. TEST UNITS

Figures 3.8 and 3.9 show the process of identifying callback-related faults for each kind of operation.

**Feature 4.3:** For each action rule, does the belief post the event specified in the rule when and only when the operation specified succeeds?

There are two associated situations relevant to this feature. First, the event specified in the rule should be posted after the operation specified in the rule is successfully applied. Second, if the operation fails, the event specified should not be posted. Hence we specify two fault types describing the failures of these two situations respectively, as shown in Table 3.4.

The fault type \textit{FT\_BE\_NOT\_POST} is level-3 in that the posting of the event in the implemented belief may be conditional. Such a condition, if it exists, is not specified in the design and may not be covered by the test cases generated.

The fault type \textit{FT\_BE\_WRONG\_POST} is level-2 because the event specified should be posted only when the operation succeeds.
3.2.5 Concurrency Discussion

Agent systems are a kind of concurrent multi-thread program because components in an agent system can be executed concurrently. Hence concurrency faults can also arise in agent systems. Much work has been done to study possible faults in concurrent software systems, such as dead-locks caused by different components accessing the same resource (e.g. a global variable), or nondeterminism resulting from interleaving concurrent events [Sen and Agha, 2007].

There also have been strategies and approaches developed for testing concurrent systems [Eytani et al., 2008]. However, agent systems have a special semantics different from other concurrent systems, such as selection of applicable plans for goal achievement, events posted by plans, and beliefs storing agent’s knowledge. Agent semantics leads to special kinds of possible software faults as presented in chapter 3. Since our research focuses on detection of those agent specific faults we do not explicitly take into account concurrent issues.

Further, our testing framework only test units within an agent and does not study interactions between agents, where concurrency issues are most likely to exist. Within an agent a plan may execute its sub-plans concurrently using parallel operators (e.g. “@post()” in JACK). The existing approaches of concurrent testing mentioned above [Eytani et al., 2008]...
may be used to detect such concurrency faults but that is not in the scope of our research.

**Summary**

This chapter introduced the test units of plans, events and beliefs within the Prometheus design paradigm, and discusses in detail test features and fault types for each of these units. These are the basis on which test cases and the testing framework are designed and implemented for our automated unit testing of an agent based system. The following chapters discuss the details of the automated testing framework that has been developed and the testing process for different types of units.
Chapter 4

Testing Framework

In our testing framework a testing tool has been developed that automatically unit tests an agent based system. This tool focuses on testing internal units within every agent. Units within an agent are tested in an appropriate order, as dependencies may exist between the units. For example, a plan may partially depend on its sub-plans because the failure of its sub-plans could cause the failure of the plan itself. A unit which has dependencies on other units should be tested after those units have been tested. Hence the order of testing of these units should be determined. We have developed an algorithm that determines such an order according to the dependencies between units.

For each individual unit, the testing tool automatically instantiates an appropriate test harness, which will then be executed to test the unit. During the process of testing a unit, a test harness generates test cases, executes test cases and analyses test outcomes. Various test harness have been designed for different types of units (plans, events and beliefs), as they have different testable features to be verified and different types of faults to be detected (refer to chapter 3). Each test harness is implemented by the testing tool via a code augmentation process, in which the implemented code of the system under test is augmented and components of the test harness are embedded. The design of different test harnesses and the details of code augmentation are also important parts in our automated testing framework.

Figure 4.1 shows the main components (or overview structure) of the testing framework. The testing tool is the controlling center of the framework and has the following functions:

- accesses the design of the system to extract information for the purpose of determining the testing order and the generation of test harnesses,
CHAPTER 4. TESTING FRAMEWORK

Figure 4.1: Main Components of The Testing Framework

- augments the code of the system for the generation of test harnesses,
- generates and executes test harnesses,
- and generates a test report that outlines testing results.

In this chapter, we first introduce in section 4.1 the global testing process that is carried out by the testing tool to test an agent system. Then, three important aspects of the testing framework are discussed in the following two sections: the order in which all units within an agent are tested (section 4.2); various test harnesses designed for testing different types of units (section 4.3) and the code augmentation process in which a test harness is automatically created (section 4.4). The final section presents the format of the test report of a system under test. The generation of test cases will be discussed in the next chapter.

4.1 Testing Process Overview

The testing tool performs an automated process for testing an agent system, as outlined in Figure 4.2. The main steps in the testing process are as follows:

- **Step 1:** When the testing process starts, the testing tool accesses the Prometheus design documentation of the system and extracts the list of all agents. These agents are then tested in alphabetical order.

- **Step 2:** For each agent, the testing tool checks each belief in the agent to verify if the data fields of the belief are correctly implemented as specified in the design descriptor.
of the belief (refer to testable feature 4.1 in section 3.2.4). We will discuss later why such verifications are carried out before all units in the agent are tested, in the part of Belief Dependency in section 4.2.1.

- **Step 3:** For each agent, the testing tool identifies the test units within the agent (plans, events and beliefs), and determines the order in which these units are tested (see section 4.2 for details).
• **Step 4:** For each unit, the testing tool implements a test harness via a process of code augmentation. The details regarding test harnesses of different types of units are discussed in section 4.3. Code augmentation is discussed in section 4.4.

• **Step 5:** The testing tool then executes the test harness to perform a unit testing process for testing the associated unit. In a unit testing process, the test harness initialises the runtime environment that is needed for the execution of test cases, generates and executes a set of test cases, collects test outcomes and compares them against the information extracted from the design documentation and identifies faults if they exist. (see section 4.3.1 for the structure of a test harness for performing a unit testing process and section 4.3.2 for details of a unit testing process).

The test harness also executes initialisation procedures that are specified by the user and are usually in the form of functions that can be invoked, for the initialisation of the runtime environment for a unit testing process, as discussed in section 4.3.3.

• **Step 6:** If any fault discussed in section 3.2 is detected in the unit testing process, one of the following three actions will be performed depending on the different levels of the fault:

  – **Action 1:**
    If a level-1 fault (an exception) occurs, the testing process will be terminated so that the user can fix the fault before the system is tested again.

  – **Action 2:**
    If a level-2 fault (an error) occurs, the testing tool will skip all untested units that are dependent on the current unit and continue testing other units.

  – **Action 3:**
    If a fault of level-3 (a warning) occurs, the testing tool will check for a pause option. There is a “Pause to accept user defined test cases on warning” provided by the testing tool which may be set by the user before the testing process starts. If the pause option is not selected, the testing tool will go on with testing the next unit. The user can investigate the fault at the end of the testing process. If the pause option is selected, the testing process will be paused and the user can investigate the fault immediately, performing one of the following:

    * **Action 3.1:** Indicate that this fault can be ignored.
CHAPTER 4. TESTING FRAMEWORK

* **Action 3.2**: Stop the testing process and fix the fault, then test the system again from the start.

* **Action 3.3**: Add additional test cases and then retest the system from the current unit, which will be tested again with user specified test cases, in addition to automatically generated test cases. User specified test cases will be introduced in chapter 5.

- **Step 7**: At the end of the testing process, the testing tool generates a test report in HTML format. The report indicates all the details of the testing process and the test results of all the units that have been tested (see section 4.5).

    Compared with other existing techniques for testing agent systems (see section 2.5.4), the testing tool in our testing framework performs a complete automated process for unit testing an agent system. In the process, user intervention is not mandatory as test harnesses are automatically implemented and each test harness is automatically executed to test the associated unit. User intervention is only necessary in some preparatory tasks for the automated testing process, such as definition of input variables (see section 5.1) and specification of initialisation procedures (see section 4.3.3). The user can also manually add additional test cases if necessary. The details on such preparatory tasks will be discussed in chapter 5.

## 4.2 Order of Testing

The units within an agent are tested in an appropriate order, which is determined by dependencies between the units. A unit should be tested after all units it depends on have been tested, because a fault cannot be assigned to the unit under test unless we know that units on which it depends perform correctly. The success of testing a unit may be partly determined by the success of testing the units on which it depends.

Dependencies between units are commonly due to design relationships between units, which are represented in the design specification of the system (i.e. a design diagram that shows the internal structure of an agent). For example, the relationship that a plan sends an event to trigger a sub-plan should be specified as a part of the agent’s structure in a design diagram, and such a relationship reflects the dependency of the plan on its sub-plan.

In this section, we introduce dependencies between units and summarise the constraints that must be satisfied relevant to each type of dependency. Then we discuss an algorithm
that has been developed to determine the order in which all units within an agent are tested, taking these constraints into account.

### 4.2.1 Dependencies between Units

We have summarised five kinds of dependencies between units as follows:

**Plan Dependency**

The success of a plan partially depends on the sub-plans triggered by the subtasks of the plan. The failure of such sub-plans may lead to the failure of the associated subtask and consequently the failure of the parent plan. A sub-plan may also run some steps that are necessary for the execution of its parent plan. For example, a sub-plan may assign the value of a variable that will later be used in the parent plan. In addition to dependencies indicated in the structural design diagram, it is possible for the tester to specify additional dependency relationships between plans. A dependee plan is considered as a sub-plan of its depender plan. Hence sub-plans and the subtask events posted by the plan should be included in the environment in which the parent plan is tested. These sub-units (sub-plans and subtask events) should have already been tested. If the sub-units have been tested successfully and a fault is detected when the parent plan is tested, we can isolate the fault to be caused by the parent unit itself.

**Cyclic Dependency**

We have introduced plan cycles in section 3.2.2. A plan cycle is considered as a single entity to be tested, as the plans within a cycle are dependent on each other. A plan within a cycle is also dependent on its sub-units (sub-plans and subtask events) which are not within the cycle. Hence the plan cycle should be tested after such sub-units have been tested. Similarly, a plan within a cycle may have parent plans which are not within the cycle. These parent plans should be tested after the associated plan cycle has been tested. Hence we specify that a plan cycle should be tested before its parent plans and after its sub-units.

For the example in Figure 4.3, the plan cycle (shaded) should be tested after $e_4$ (a subtask event of $P_3$), $e_5$ (a subtask event of $P_1$), $e_6$ (a subtask event of $P_2$), $P_41$ and $P_42$ (sub-plans of $P_3$), $P_5$ (a sub-plan of $P_1$) and $P_6$ (a sub-plan of $P_2$) have been tested. The cycle should be tested before $P_0$ ($P_1$ is its sub-plan.) and $P_7$ ($P_2$ is its sub-plan) are tested.
CHAPTER 4. TESTING FRAMEWORK

Figure 4.3: Example of Cyclic Structure

Event Dependency

An event is tested to check the features of overlap (if the event has multiple applicable plans in some situations) and coverage (whether there is at least one applicable plan in every situation), by checking the values of the context conditions of the plans that handle the event. Hence before an event is tested, it is necessary to verify that the context conditions of all plans that handle the event have been correctly implemented. Those plans must be tested before the events that they handle.

Belief Dependency

Each action rule of a belief should be tested to verify if the rule works properly as discussed in section 3.2.4. An action rule in Prometheus is specified as a tuple \(<operation, event>\), in which the event is posted by the belief once the operation has been successfully applied to the belief (see section 2.3 for the structure of an action rule). If the event specified in an action rule is not handled by applicable plans as expected, the action rule will not work properly. Hence any event specified in the action rules of the belief under test should have been tested before the belief itself it tested. For example, \(B1\) in Figure 4.4 should be tested after \(e2\) and \(e3\).

The plans that access (read or change) a belief also depend on the belief. For example, a plan may read a data field of the belief. If such a data field is not correctly implemented as specified in the design documentation (e.g., wrong type), the plan may fail when it accesses the data field. Therefore, before the plans that access a belief are tested, it is necessary
to verify that the data fields of the belief are correctly implemented as specified. However, such a verification is a code checking process, in which the implemented code of the belief is checked to confirm that the data fields are implemented as specified in the design. The verification does not require the creation and execution of a test harness. Furthermore, such a verification for every belief is independent and does not depend on any other unit. Therefore, the verification of data fields for all beliefs is performed before the order of testing is determined and all units are tested, and we do not consider this kind of dependency when determining the order of testing (refer to Step 2 in the global testing process shown in Figure 4.2).

Hidden Dependencies

In addition, the developer is also allowed to directly specify some particular unit dependencies. For example, Plan\(_1\) may check the state of a belief and be pending until Plan\(_2\) updates the belief to a particular state. The developer can specify that Plan\(_1\) depends on Plan\(_2\). We call such a dependency a hidden dependency as it is not explicitly represented in the structure of an agent. If the developer specifies a hidden dependency between two units, the unit on which the other unit depends is considered as a sub-unit of the other one when the order of testing is determined.

Summary

In conclusion, we specify five constraints that are associated with unit dependencies and affect the order in which units are tested.

- Con-1: A plan should be tested after each of its sub-units (subtask events and sub-plans) have been tested.
CHAPTER 4. TESTING FRAMEWORK

Figure 4.5: Example for Determining Order of Testing

- Con-2: A plan cycle should be tested after all its sub-units have been tested.
- Con-3: A plan cycle should be tested before all its parent plans are tested.
- Con-4: An event should be tested after all the plans that handle it have been tested.
- Con-5: A belief should be tested after each event specified in the action rules of the belief has been tested.

Using these constraints, we have developed an algorithm that determines the order of testing all units within an agent, as represented in the next subsection.

4.2.2 Algorithm

The algorithm specifies a process that consists of six steps as presented below. A test queue will be generated in the process. We use the example of Figure 4.5 to illustrate the process and abbreviate the following: Plan Test - PT; Event Test - ET; Belief Test - BT; Plan Cycle Test - CT. We also explain how every constraint summarised above is satisfied by the algorithm.

- **Step 1: Identify Root Plans**
  A root plan handles a percept or an external message from outside the agent. No units within the same agent depend on a root plan, such as $P_0$, $P_{81}$ and $P_{82}$ in Figure 4.5. Every root plan should be identified so that other units it directly or indirectly depends on can be explored and the order in which they are tested can be determined.
PROCEDURE getOrderP(PlanNode P) // explore a plan and its descendent units
    IF isInTestQueues(P) THEN terminate the procedure
    stack.push(P) // “stack” stores the current path explored
    FOR EACH sub-plan P_i of P
        IF P_i is the ancestor of any Plan in the stack THEN
            //add a plan cycle unit to the test queue (TQ)
            TQ.add(CT(P_i and all its descendant plans in the stack))
        ELSE getOrderP(P_i) //explore a sub-plan
        FOR EACH outgoing event of P e
            TQ.add(ET(e)) //add an event unit to the test queue
        FOR EACH belief B_i that is accessed by P
            getOrderB(B_i)
    TQ.add(PT(P)) // add the current plan to the test queue
    stack.pop(P) // P and all its descendent units have been explored, so P is removed from the stack
END PROCEDURE

PROCEDURE getOrderB(BeliefNode B) // explore a belief and its descendent units
    IF isInTestQueues(B) THEN terminate the procedure
    FOR EACH action rule AC_i of B
        FOR EACH Plan P_j that handles the event E_i specified in AC_i
            getOrderP(P_j) //explore a sub-plan of the belief
        IF E_i is handled by multiple plans (P_j, P_k, . . .) THEN
            TQ.add(ET(E_i)) // add an event unit to the test queue
    TQ.add(BT(B)) //add the current belief to the test queue
END PROCEDURE

// check whether the given unit is in any test queue that has been generated for root plans
FUNCTION isInTestQueues(UnitNode N) RETURNS boolean // N can be any kind of unit
    IF N has been added in any test queue that has been generated THEN
        RETURN true
    ELSE
        RETURN false
END FUNCTION

Figure 4.6: Pseudo Code for Step 2 of Testing Order Determination

- **Step 2: Generate The Draft Test Queues**

For each root plan in the agent under test, we perform a procedure of modified depth-first search as outlined in Figure 4.6 (the procedure getOrderP) and one draft test queue is generated. These test queues (if there are multiple root plans), will be combined in the following steps, to finally generate one single test queue. The “getOrderP” procedure is invoked with a root plan as the argument. This procedure realises a typical depth-first search but also identifies cyclic dependencies, as well as plans that handle the same event. The “getOrderB” procedure is also invoked to explore any belief accessed by a plan. A test queue will be created for each root plan. There are three root plans in
Figure 4.5 so they are explored respectively, and three draft test queues are created, each of which ends with the associated root plan, as shown in Figure 4.7.

- \( TQ_1: \) PT(P5), ET(e5), CT(P3, P1, P2), PT(P41), PT(P42), ET(e4), PT(P3), ET(e3), CT(P6, P3, P1, P2), PT(P6), ET(e6), PT(P2), ET(e2), PT(P1), ET(e1), PT(P7), ET(e7), PT(P0)

- \( TQ_2: \) PT(P10), ET(e10), PT(P9), ET(e9), BT(B1), PT(P81)

- \( TQ_3: \) PT(P10), ET(e10), PT(P11), ET(e11), PT(P12), ET(e12), PT(P82)

Figure 4.7: Draft Test Queues

In cases where a unit is a sub-unit of multiple units, this unit will be added into multiple test queues (i.e. P10 and e10 in \( TQ_2 \) and \( TQ_3 \)). Such duplicate items will later be removed from test queues in Step 6.

In addition, if the sub-units of a plan or a belief are explored in different orders, the test queues generated may be different. For example, if \( P12 \) is explored before \( P11 \) in the generation of \( TQ_3 \), the queue will be:

- \( TQ_3: \) PT(P12), ET(e12), PT(P10), ET(e10), PT(P11), ET(e11), PT(P82)

However, this does not affect the correctness of the algorithm, as all constraints are still satisfied.

We use the test queues generated in Figure 4.7 as the example in the following steps.

- **Step 3: Eliminate The Units within A Plan Cycle**

In the test queues generated in the above step, plans and events that are within a plan cycle are also added as individual test items, (i.e. PT(P6), ET(e6), PT(P3), ET(e3), PT(P1), ET(e1), PT(P2), ET(e2) in \( TQ_1 \) (see Figure 4.7), and thus should be removed from the test queue as they will be tested when the associated plan cycles are tested. For example, PT(P6), ET(e6), PT(P3), ET(e3), PT(P1), ET(e1), PT(P2) and ET(e2) are redundant as they occur in cycles CT(P3, P1, P2) and CT(P6, P3, P1, P2). There are no plan cycles in \( TQ_2 \) and \( TQ_3 \), which are therefore not modified (see Figure 4.8).

- **Step 4: Relocate Plan Cycles**

A plan cycle in a test queue may not satisfy Con-2 or Con-3. In other words, the cycle may not be after all its sub-units or before all its parent plans in the test queue. In \( TQ_1 \) in Figure 4.8, CT(P3, P1, P2) is before ET(e4), while e4 is a subtask event posted
CHAPTER 4. TESTING FRAMEWORK

- **TQ₁**: PT(P5), ET(e5), CT(P3, P1, P2), PT(P4₁), PT(P4₂), ET(e4), CT(P6, P3, P1, P2), PT(P7), ET(e7), PT(P0)
- **TQ₂**: PT(P10), ET(e10), PT(P9), ET(e9), BT(B1), PT(P8₁)
- **TQ₃**: PT(P10), ET(e10), PT(P1₁), ET(e1₁), PT(P1₂), ET(e1₂), PT(P8₂)

*Figure 4.8: Test Queues after Removing Plans in Cycles*

by P3. In this step, each cycle in every test queue is checked and is relocated to satisfy Con-2 and Con-3 if it does not already do so. Figure 4.9 shows the result of this step. CT(P3, P1, P2) in TQ₁ is relocated. TQ₂ and TQ₃ are still not modified.

- **TQ₁**: PT(P5), ET(e5), PT(P4₁), PT(P4₂), ET(e4), CT(P6, P3, P1, P2), PT(P7), ET(e7), PT(P0)
- **TQ₂**: PT(P10), ET(e10), PT(P9), ET(e9), BT(B1), PT(P8₁)
- **TQ₃**: PT(P10), ET(e10), PT(P1₁), ET(e1₁), PT(P1₂), ET(e1₂), PT(P8₂)

*Figure 4.9: Test Queues after Relocating Cycles*

- **Step 5: Combine Plan Cycles**
  Plan cycles that overlap should be combined. By overlap we mean plan cycles that have at least one plan in common. In the example CT(P3, P1, P2) is a sub-set of CT(P6, P3, P1, P2) and should be merged (see Figure 4.10).

- **TQ₁**: PT(P5), ET(e5), PT(P4₁), PT(P4₂), ET(e4), CT(P6, P3, P1, P2), PT(P7), ET(e7), PT(P0)
- **TQ₂**: PT(P10), ET(e10), PT(P9), ET(e9), BT(B1), PT(P8₁)
- **TQ₃**: PT(P10), ET(e10), PT(P1₁), ET(e1₁), PT(P1₂), ET(e1₂), PT(P8₂)

*Figure 4.10: Test Queues after Combining Cycles*

- **Step 6: Combine test queues**
  All test queues are combined to a global test queue following the four sub-steps:
– **Step 6.1: Combine the test queues whose root plans handle the same event**

There may be some test queues whose root plans handle the same event \( e \), which is a percept or an external message. Such an event should also be tested, but it is not added in any test queue yet. In this step these test queues, if they exist, are concatenated to form a new test queue, in which a new event test item \( ET(e) \) is added as the last item. The order in which these queues are combined is random, as there are no dependencies between units in different test queues. \( ET(e) \) is added as the last item because it should be tested after all the root plans that handle it (refer to Con-4).

In Figure 4.10, \( TQ_2 \) and \( TQ_3 \) are combined because their root plans \( P81 \) and \( P82 \) handle the same event \( e81 \), which is an external message from outside the agent under test. Consequently a new \( TQ_2 \) is obtained with \( ET(e81) \) as the last item (see Figure 4.11).

| \( TQ_1 \): \( PT(P5), ET(e5), PT(P41), PT(P42), ET(e4), CT(P6, P3, P1, P2), PT(P7), ET(e7), PT(P0) \) |
| \( TQ_2 \): \( PT(P10), ET(e10), PT(P9), ET(e9), BT(B1), PT(P81), PT(P10), ET(e10), PT(P11), ET(e11), PT(P12), ET(e12), PT(P82), ET(e81) \) |

*Figure 4.11: Test Queues after Combination*

– **Step 6.2: Add triggering events of root plans**

If a test queue is not combined in Step 6.2, a new event test item of the triggering event of the queue’s root plan is added into the queue as the last item (i.e. \( ET(e0) \) in Figure 4.12).

| \( TQ_1 \): \( PT(P5), ET(e5), PT(P41), PT(P42), ET(e4), CT(P6, P3, P1, P2), PT(P7), ET(e7), PT(P0), ET(e0) \) |
| \( TQ_2 \): \( PT(P10), ET(e10), PT(P9), ET(e9), BT(B1), PT(P81), PT(P10), ET(e10), PT(P11), ET(e11), PT(P12), ET(e12), PT(P82), ET(e81) \) |

*Figure 4.12: Test Queues after Adding Triggering Events of Root Plans*
– Step 6.3: Combine all test queues to a global one

All test queues are combined together to a global test queue. The order of combination is unimportant as there are no dependencies between units in different test queues. (see Figure 4.13).

* $T Q_1$: $PT(P5)$, $ET(e5)$, $PT(P41)$, $PT(P42)$, $ET(e4)$, $CT(P6, P3, P1, P2)$, $PT(P7)$, $ET(e7)$, $PT(P0)$, $ET(e0)$, $PT(P10)$, $ET(e10)$, $PT(P9)$, $ET(e9)$, $BT(B1)$, $PT(P81)$, $PT(P10)$, $ET(e10)$, $PT(P11)$, $ET(e11)$, $PT(P12)$, $ET(e12)$, $PT(P82)$, $ET(e81)$

Figure 4.13: Combination of All Test Queues

– Step 6.4: Remove duplicate items

For each unit that has duplicate items in the test queue (i.e. two $PT(P10)$ and two $ET(e10)$ in Figure 4.13), only the first appearance of the duplicate items remains in the test queue (i.e. the $PT(P10)$ and $ET(e10)$ between $ET(e0)$ and $PT(P9)$). Others will be removed in order to guarantee that all other units that depend on the unit are after that unit in the final test queue. For instance, $e10$ and $P10$ are sub-units of both $P81$ and $P11$ and should be prior to both of them in the final test queue. Hence only the first $ET(e10)$ and $PT(P10)$ (between $ET(e0)$ and $PT(P9)$) remain and the others are removed. Consequently, the final test queue is generated as shown in Figure 4.14.

* $T Q_1$: $PT(P5)$, $ET(e5)$, $PT(P41)$, $PT(P42)$, $ET(e4)$, $CT(P6, P3, P1, P2)$, $PT(P7)$, $ET(e7)$, $PT(P0)$, $ET(e0)$, $PT(P10)$, $ET(e10)$, $PT(P9)$, $ET(e9)$, $BT(B1)$, $PT(P81)$, $PT(P11)$, $ET(e11)$, $PT(P12)$, $ET(e12)$, $PT(P82)$, $ET(e81)$

Figure 4.14: Final Test Queue

In conclusion, the algorithm we have developed determines the order of testing for units within an agent, with respect to dependencies between units. The product of the algorithm is a global test queue that includes all identified units of an agent. These units can then be tested one by one following the order in the queue. Test harnesses, which automatically test units and are automatically implemented by the test program, are introduced in the next section.
4.3 Testing Harness

Test harnesses have been developed in the testing framework for testing different types of units: *plans*, *events* and *beliefs*. A test harness consists of a series of components that automatically verify testable features and detect faults as discussed in chapter 3, such as a test agent that initialises the testing environment and executes test cases and testing messages for the communication between the test agent and the agent under test. In a test harness, test code is embedded into the unit under test to track the runtime behaviour and states of the unit. A test harness may have mock agents that simulate other agents that interact with the agent under test in the system. The structure of test harnesses for different types of units are discussed in the following (section 4.3.1), together with the unit testing process performed by them (section 4.3.2).

The test harness tests an individual unit. Prior to executing each test case for that unit, it may be necessary to perform some initialisation procedures such as the setting up of connections to external servers, the population of databases, the initialisation of global variables and so on. The user should implement such procedures (if they are not yet implemented) and declare them in the test descriptor of the unit \(^1\) in order for the procedures to be invoked by the associated test harness before the unit is tested. The details regarding implementation and declaration of initialisation procedures are discussed in section 4.3.3.

The test harness of the unit under test is automatically implemented by the testing tool via a process of code augmentation. During such a process the implemented code of the system is augmented by embedding the code of test harness components into the system. We use JACK [Winikoff, 2005] as the implementation platform, but the principles are also general to other implementation platforms for agent systems. The details of the code augmentation will be discussed in section 4.4.

4.3.1 Structure of Test Harnesses

Figure 4.15 shows an abstract view of a test harness. In general, a test harness always consists of two parts as follows:

- **test-driver part**
  
  The test-driver part of a test harness initialises the testing environment for the unit

\(^1\)The test descriptor of a unit is an extension of the unit’s design descriptor, specifying testing specific information, see page 89+3.
CHAPTER 4. TESTING FRAMEWORK

under test, activates the unit testing process, collects test outcomes and analyses the testing results.

- subsystem under test

  The subsystem under test consists of the unit under test and other relevant components that are necessary for the execution of the unit under test. Test code, which tracks runtime states and behaviour of the unit under test, is inserted into the subsystem under test.

A test harness may have mock agents that simulate other agents that interact with the agent under test. The unit under test may interact with external programs during its execution. Such an external program can be an external system, such as a server to which the unit under test connects; or can be an external agent in the same system, with which the unit under test communicates following protocols. If the unit under test interacts with external systems, the user is required to start up these systems before the unit is tested. For external agents with which the unit under test interacts, the testing tool may automatically create mock agents to simulate them in the associated test harness. This is because these agents may not be implemented or completely tested when the unit is tested.

The implementation of mock agents is not mandatory. If an external agent that communicates with the subsystem under test has been implemented and tested sufficiently, the user can specify that the agent is directly used in which case a mock agent is not generated for the replacement.

The details of the parts of a test harness are discussed as follows:

Figure 4.15: Abstract View of A Test Harness
CHAPTER 4. TESTING FRAMEWORK

Test-Driver

The test-driver contains a test agent, a test-driver plan that is embedded into the agent under test and a series of testing specific message-events that are sent to and from the test agent to communicate with the agent under test. The test agent in a test harness activates a unit testing process, receives result messages sent by the agent under test that log test outcomes and analyses test outcomes. The test-driver plan is developed in order to directly interact with the unit under test. Often the unit under test cannot be activated from outside the agent under test. For example, a plan under test may handle an event that is an internal event of the agent. Therefore a test-driver plan needs to be embedded into the agent under test in order to communicate with the unit under test. The test-driver plan is triggered by an activation message that is sent by the test agent. When a test case finishes, the test-driver plan sends a finished message to the test agent.

Subsystem under Test

The subsystem under test is the portion of the system that is necessary for testing the relevant unit. It includes the key units and the supporting hierarchy of the unit under test. The former are the unit under test and other relevant units whose runtime behaviours or states should be tracked and logged for testing purposes. While the latter are the units on which the unit under test depends for full execution. Key units and the supporting hierarchy for different types of units are summarised as follows, as well as how they work for testing a unit:

- **plan**: The key units are the plan itself and its triggering event. The supporting hierarchy consists of the beliefs accessed by the plan, the descendent plans of the plan under test and the events that trigger these plans (see Figure 4.16).

  The test-driver plan performs a unit testing process by posting the triggering event of the plan under test. The plan may or may not be triggered. This is a testable feature to be verified (Feature 1.1 in section 3.2.1). If the plan is triggered and executed, its runtime states and behaviour will be tracked and other testable features will be verified. The details of the unit testing process are discussed in section 4.3.2.

- **plan cycle**: The key units are all plans in the cycle and their triggering events. The supporting hierarchy consists of the beliefs accessed by the cycle, the descendent plans of the cycle and the events that trigger these descendent plans.
For testing a plan cycle, a plan within the cycle (cyclic plan in short) is randomly chosen as the start point, and the test-driver plan posts the triggering event of this plan. All cyclic plans are tested from the chosen plan in the order that they are located in the cycle. A cyclic plan is tested to verify both plan specific features (refer to section 3.2.1) and cyclic specific features (refer to section 3.2.2). Figure 4.17 shows the subsystem under test for just one plan of the cycle (Plan.1). This activation however must be done in turn for each plan of the cycle.
• **event:** The *key units* are the event and all the plans that handle that event. There is no supporting hierarchy for an event because only the applicability of the plans that handle the event is considered and the full execution of these plans is not required (see Figure 4.18). Test code is inserted into each of these plans to ensure that only the context condition of each plan is evaluated and other parts of the plan are not executed. The details of inserting such test code will be discussed in section 4.4.

The test-driver plan posts the event under test. After that the context condition of each plan that handles the event under test is tracked, to evaluate the applicability of these plans for the event under test.

• **belief:** The *key units* are the belief and the events specified in the action rules of the belief. There is no supporting hierarchy for a belief as we check only the posting of the events specified in the action rules of the belief and the execution of the plans that handle those events are not required (see Figure 4.19).
A belief is tested to verify if its data fields are correctly implemented, if appropriate methods are implemented for each action rule, and if for each action rule the belief posts the event specified in the rule when and only when the operation specified succeeds (refer to Feature 4.1 - 4.3 defined in section 3.2.4). The first two features are verified in the testing framework by checking the source code of the belief under test, and comparing the implemented data fields and action rules against the information specified in the belief’s design descriptor. This is a static test and does not require tracking the runtime behaviour and states of the belief under test. Section 3.2.4 has discussed how to verify these two features.

The test harness developed for a belief therefore only verifies that the appropriate events are posted when and only when particular kind of belief operations occur. A belief operation is one of insert, update, delete and modify. Insert, update or delete is a concrete operation while modify is the generalisation of these three concrete operation. An action rule with the modify operation indicates that any of insert, update and delete should activate the event specified in the rule.

The test-driver plan extracts the action rules of the belief under test from the design specification and performs the operation specified in each rule in the case of insert, update or delete. Then possible posting of events and success of the operation are tracked by the test code inserted in order to identify possible faults.

If the operation specified in an action rule is modify, the rule will be decomposed to be three sub action rules, which respectively specify the operations of insert, update and delete, with the same event to be posted as the parent rule. Each sub action rule will be respectively verified, then test outcomes of these three sub action rules will be analysed together as the result of verifying their parent action rule.

### 4.3.2 Unit Testing Process

The unit testing process performed by a test harness consists of eight general steps as follows:

- **Step 1**: The test agent initialises the runtime environment.

  A unit testing process is activated by the test agent, which initialises the runtime environment that is required for the execution of test cases. The agent under test and the unit under test are instantiated. If the developer has declared some initialisation
procedures in the design for the unit under test, such procedures are also executed (see section 4.3.3).

- **Step 2**: The test agent generates test cases.
The test agent then generates a set of test cases. The inputs of test cases are the value combinations of the input variables of the unit. The generation of test case inputs will be discussed in chapter 5.

- **Step 3**: The test agent triggers the test-driver plan.
For each test case, the test agent sends an activation message, which carries the test inputs of the test case to trigger the test-driver plan.

- **Step 4**: The test-driver plan sets up the inputs.
When the test-driver plan is triggered by the activation message, it will set up the inputs by assigning input values to relevant variables. We will introduce in chapter 5 how input values of a test case are assigned to associated variables at runtime. The context condition of the test-driver plan is always true as the plan is required to be always applicable in order for the testing to execute.

- **Step 5**: The unit under test is activated.
After setting up the inputs of the current test case, the test-driver plan activates the execution of the unit under test in different ways, according to different types of units, as mentioned in section 4.3.1: for testing a plan, the test-driver plan posts the triggering event of the plan under test; for testing an event, the test-driver plan posts the event under test; for testing an action rule of a belief, the test-driver plan performs the operation specified in the action rule.

- **Step 6**: The subsystem under test is tracked.
The test code that has been embedded into the subsystem under test tracks the runtime behaviours and internal states of the unit under test and the other key units, in order to detect the faults as described in chapter 3. The information tracked is reported on by the test code to the test agent by sending messages. Details regarding how test code is inserted and what information is tracked are discussed in section 4.4.

- **Step 7**: After the subsystem under test is executed and tracked, the test-driver sends a *Finished Message* (see Figure 4.15) to the test agent to end the current test case.
Then the test agent runs the next test case by sending another activation message to
the agent under test and Step 3 to Step 7 will be repeated.

• **Step 8**: The test agent analyses test outcomes.
  After the last test case is executed, the test-driver plan sends a *Finished Message* to
  the test agent. The test agent then analyses the test outcomes collected. The analysis
  results are saved for the later generation of the test report (see Step 7 in Figure 4.2).

### 4.3.3 Initialisation Procedures

Initialisation procedures for a unit have different scopes according to how widely procedures
are used by units in the system and different levels depending on when the procedures will
be executed. They are discussed in detail as follows:

• **Scope**
  Some initialisation procedures are specific to a unit, while others are general to most or
  all units within the same agent. For example, a procedure may initialise agent variables
  that are necessary for most or all units in the agent. Similarly, there are initialisation
  procedures that are general to most or all units in the system. Below is an example
  showing the initialisation procedures in different scopes required for testing the plan
  “BuyBooks”, which purchases books in an electronic bookstore.

  1. First, some global variables, such as the book categories, need to be initialised.
     These global variables may also necessary to the execution of most or all of the
     other units in the system.
  2. Second, the plan belongs to the agent “StockAgent”, which populates the books
     database when it is initialised. The data in the books database is also necessary
     for the other units in this agent.
  3. Lastly, the plan must connect to an electronic payment server before processing
     the purchase, so a connection to the server would need to be established before
     this plan is executed. This initialisation is specific to the particular “BuyBooks”
     plan, the individual unit.

• **Level**
  The level of an initialisation procedure determines when the procedure will be executed.
  We define three levels for initialisation procedures as follows:
CHAPTER 4. TESTING FRAMEWORK

1. “system”: The procedure is relevant to the initialisation of global properties, such as the connections to external servers or databases and the initialisation of global variables. The procedure should be executed at the beginning of the unit testing process.

2. “agent”: The procedure is relevant to the initialisation of the agent under test, such as initialising agent properties that are required for the execution of the agent under test, and should be executed after the agent under test is instantiated.

3. “unit”: The procedure is relevant to the initialisation of the unit under test, such as initialising some unit properties that are required for the execution of the unit under test, and should be executed after the unit under test is instantiated.

A system-level procedure may not be general to most or all units in the system. Instead, it may be specific to the units within an agent, or to a certain unit. Also, an agent-level procedure can be specific to a unit, or be general to most or all units in the system; a unit-level procedure may be general to most or all units in an agent, or in the system. This is because the level of a procedure indicates when the procedure will be executed, while the scope of the procedure describes where it is required. For example, the plan “BuyBooks” may require some particular data of the providers database, which should be populated once the “StockAgent” is instantiated. The developer could implement an “initProviderDB()” method whose scope is the “BuyBooks” plan. The method is at the agent-level because the procedure is associated with the initialisation of the agent.

Before a unit is tested, such initialisation procedures should have been implemented by the developer in the form of methods that can be invoked by the associated unit test harness. Usually such methods have existed in the implementation as they are necessary for the execution of the unit. If not, they should be implemented before the unit is tested. In the example of the “BuyBooks” plan mentioned above, three procedures can be implemented with three different scopes: “initGlobalVariables()” which is a global method of the system and is general to most or all units in the system, “initBookDB()” which is a member method of the agent “StockAgent” and is general to most or all units in the agent, and “initElecPayment()” which is a member method of the plan “BuyBooks”. The method establishes a connection to an electronic payment as mentioned above and is specific to the plan.

There are two guidelines for the implementation of an initialisation procedure. First, methods that exist in the implementation should be used where possible, particularly if such
methods have been tested using traditional testing techniques such as JUNIT\(^2\). Implementing new methods increases the cost of testing a unit and also increases the risk that mistakes may exist in the newly implemented code. Second, any newly implemented procedure should only contain simple operations, such as establishing a connection to a server, or inserting some particular data to a database.

**Test Descriptors**

In order for a unit test harness to perform initialisation procedures for the unit, the developer should declare them in the *test descriptor* of the unit itself, the agent that contains the unit or the system. A *test descriptor* is an extension to the Prometheus design descriptor of an entity (an agent or a unit), capturing testing specific properties of the entity that are in addition to the usual design descriptor that specifies the properties of the entity. The developer can also specify a *system test descriptor* for the system, describing testing specific information that is general to the system. The system test descriptor contains the declaration of the procedures that are necessary for the initialisation of most or all units in the system (e.g. the “initGlobalVariables()” method mentioned above). The procedures that are general to most or all units within an agent are declared in the relevant agent test descriptor, such as the “initBookDB()” that is declared in the test descriptor of the agent “StockAgent”. Those procedures that are specific to a single unit are declared in the relevant unit test descriptor, such as the “initElecPayment()” that is declared in the test descriptor of the plan “BuyBooks”. When a unit is tested, the associated test harness will respectively extract the declarations of the initialisation procedures in the system test descriptor, the relevant agent test descriptor and the unit test descriptor, then execute these procedures at the appropriate time. A test descriptor is in the form of textual description following an XML format (see the example in Figure 4.20). Multiple initialisation procedures can be specified where each is in the following format:

\[
<\text{level}, \text{owner}\_\text{object}, \text{is}\_\text{static}, \text{method}\_\text{call}, \text{comment}>.
\]

- **Level**: one of “system”, “agent” or “envet”, as discussed at the beginning of this section
- **Test Owner\_\text{object} (owner in short)**: the object to which the procedure belongs.

As mentioned above, each initialisation procedure should be developed in the form of a method that can be invoked at runtime and should belong to an object\(^3\). With this

\(^2\)http://www.junit.org/  
\(^3\)in terms of Java programming
field, the method can be invoked by the test harness at runtime.

- **Is static**: The related method may be developed as a static method or a non-static method by the developer according to the application. This field indicates if the method is a static one or not. This field is necessary as the test harness needs to know if the method is invoked as a member method of the owner object (e.g. `ag.initDB()`) or as a static method of the class of the owner object (e.g. `StockAgent.ConnectServer()`).

- **Method call**: the invocation statement.
  It gives the name of the method, including the list of arguments if required. With this statement and the **owner** specified, the initialisation procedure can be executed by the test harness at runtime.

- **Comment**: a textual description about the functionality of the procedure.

The procedures in the same level and in the same test descriptor are executed in the order in which they are declared in the test descriptor. Table 4.1 shows examples of initialisation procedures for the plan “BuyBooks” and the order in which they are executed.
CHAPTER 4. TESTING FRAMEWORK

<table>
<thead>
<tr>
<th>level</th>
<th>owner</th>
<th>is static</th>
<th>method call</th>
<th>comment</th>
<th>declared in</th>
<th>order of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>Main</td>
<td>yes</td>
<td>initGlobalVariables()</td>
<td>initialise global variables</td>
<td>system test descriptor</td>
<td>1</td>
</tr>
<tr>
<td>agent</td>
<td>StockAgent</td>
<td>yes</td>
<td>initBookDB()</td>
<td>populate the books database</td>
<td>agent test descriptor</td>
<td>2</td>
</tr>
<tr>
<td>agent</td>
<td>StockAgent</td>
<td>yes</td>
<td>initProviderDB()</td>
<td>populate the providers database</td>
<td>unit test descriptor</td>
<td>3</td>
</tr>
<tr>
<td>unit</td>
<td>BuyBooks</td>
<td>no</td>
<td>initElecPayment()</td>
<td>sets up connection to Amazon.com</td>
<td>unit test descriptor</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.1: Examples of Initialisation Procedures

4.4 Code Augmentation

A test harness is automatically implemented via a process of code augmentation, in which two programs are separately created and executed: the test agent program, and the subsystem under test program whose code has been augmented to include the code of the test-driver plan, test code and the code of mock agents. By doing this the execution of the test agent is independent of the execution of the subsystem under test. This independence is critical because the test agent may fail to control the subsystem under test in some situations if the test agent and the subsystem under test are in the same program at the implementation level. For example, the subsystem under test may terminate at runtime due to errors or exceptions. If the test agent and the subsystem under test are in the same program (i.e. The test agent is embedded into the system under test), the whole program including the test agent will also be terminated in such a situation and the relevant unit testing process will stop unexpectedly. Therefore, a test harness is implemented as two separate programs at the implementation level.

When a particular unit is tested, the testing tool first starts the test agent program, in which test cases for that unit are generated and each test case is executed. In the execution of each test case, the subsystem under test program is newly executed by the test agent program and the associated unit is tested.

Test Agent

The code of the test agent has been manually developed for the JACK implementation platform. The test agent contains an activation plan that generates test cases and runs each

---

4We use the DCI communication mechanism (http://www.aosgrp.com/documentation/jack/AgentManualWEB/index.html) provided by JACK to implement the communication between agents at runtime in different programs.
Before code augmentation:

```java
public plan Activation
    Plan extends Plan {
    #sends event Activation_Message acti_msg;
    body() {
        // the name of the unit under test
        String unit_name = "<UNIT UNDER TEST>";

        // generate test inputs
        TestInputs test_inputs = generateTestInputs(unit_name);
        // run each test case,
        for(int i=0; i<test_inputs.size(); i++)
        {
            OneCaseInputs one_case = test_inputs.get(i);
            acti_msg = new Activation_Message(one_case);
            @send(getAgentUnderTestName(), acti_msg);
        }
    }
}
```

After code augmentation:

```java
public plan Activation
    Plan extends Plan {
    ....
    body() {
        // the name of the unit under test
        String unit_name = "BuyBooks";
        ....
    }
}
```

Figure 4.21: Code of Activation Plan in The Test Agent

test case by sending an action message to the agent under test. It also contains a series of plans that handle the results messages sent by the subsystem under test and an analysis plan that analyses test outcomes. We will discuss later in this section how the internal components of the test agent work with the test code that has been embedded into the system for testing a particular unit.

In the code augmentation process for a particular unit, the code of the test agent program is copied into a particular directory specified by the user, then the code of the test agent is updated for testing the particular unit. In the generic code, the names of the unit under test or the other relevant units in the subsystem under test are denoted by particular string keywords, which are then updated with the exact names of the associated units to make the test agent program specific to the particular unit. Figure 4.21 shows how the code of the
activation plan is updated for testing a particular plan “BuyBooks”. The keywords in the other part of the test agent code are also updated in the same way. After the updating of all these keywords, the code of the test agent program is compiled to make the program ready for execution.

**Subsystem under Test**

To implement the subsystem under test program, the original source code of the system is copied into an individual directory. Then the test-driver plan and mock agents are embedded and test code is appropriately inserted into the code of the system. Finally the new program is compiled to make the program ready for execution.

In general code augmentation can be problematic for the implementation of subsystem under test. This is because the code augmented may modify the time taken for relevant pieces of code to execute and consequently may lead to concurrency problems. However, our research does not explicitly deal with concurrency issues (see page 64). Also the test-driver plan activates the unit under test in a single thread to avoid concurrency. Therefore code augmentation is not problematic in our research.

In the augmentation process for the subsystem under test, four kinds of code are inserted:

1. A main class is added for testing purposes, which is set as the entry class of the subsystem under test program. In the class the agent under test is instantiated and the initialisation procedures at the system level and at the agent level are executed before and after the agent under test is instantiated respectively. The code of the main class is shown in Figure 4.22. In the code the keyword that denotes the name of the agent under test is updated to the particular agent name (“StockAgent” in this example), which contains the unit to be tested.

2. The code of the test-driver plan and the relevant events it handles and sends is integrated into the agent under test. Also, the generic code of the test-driver plan contains the keywords that denote the name of the unit under test and the other key units in the subsystem under test. For example, the code of the test-driver plan for testing a plan contains a keyword that denotes the type name of the triggering event of the

---

5We use the code of the whole system for code augmentation because the code of the key units and supporting hierarchy is difficult to be separated from the system with successful compiling. Hence we use a simple strategy to augment based on the code of the whole system and insert particular test code to make those units that do not belong to the subsystem under test not executable, refer to “Limitation of execution” in this section for details.
### Before code augmentation:
```java
public class Test_Main {
    public static void main(String[] args) {
        // run the initialisation procedures in the system level
        runInitProcedures(system();

        // instantiate the agent under test
        inst.agent("<AGENT_UNDER_TEST_TYPE>");

        // run the initialisation procedures in the agent level
        runInitProcedures(agent();
...
    }
}
```

### After code augmentation:
```java
public class Test_Main {
    public static void main(String[] args) {
        ....
        // instantiate the agent under test
        inst.agent("StockAgent");
        ....
    }
}
```

---

**Figure 4.22: Code of The Test Main Class for The SUT**

plan under test, because the test-driver plan needs to post out the triggering event to execute the plan under test. The code of the test-driver plan, before and after being updated, is shown in Figure 4.23. The code of the test-driver plans for testing an event and a belief is similar and is shown in appendix A.

3. Test code is embedded into the subsystem under test, which contains the unit under test and other relevant units (*supporting hierarchy* and *key units*). The test code inserted detects runtime states and behaviour of the unit under test and works with test driver components to identify the faults introduced in section 3.2. The details of what and how test code is embedded are discussed in sections 4.4.1 to 4.4.4.

4. The code of mock agents is automatically generated and is embedded into the system. The insertion of mock agents is described later in section 4.4.5.

When test code is embedded into the implemented code of the whole system, there are two aspects considered: *limitation of execution* and *test code inserted for tracking particular*
CHAPTER 4. TESTING FRAMEWORK

Before code augmentation:

```java
public plan TestDriver_Plan extends Plan {
    ....
    #handles event Activation_Message act_ev;
    #posts event <TRIGGER_EVENT> trigger;
    context() {
        true;
    }
    body() {
        //set up the test inputs
        setTestInputs(ev.getInputs());
        //run the initialisation procedures in the unit level
        runInitProcedures_unit();
        //posts the event to activate the plan under test
        @subtask(trigger);
    }
    ....
}
```

After code augmentation:

```java
public plan TestDriver_Plan extends Plan {
    ....
    #handles event Activation_Message act_ev;
    #posts event BuyBooks_Ev trigger;
    ....
}
```

Figure 4.23: Code of Test-Driver Plan for Testing A Plan

1. Limitation of Execution

In order to ensure that the runtime environment is controlled, only the units that belong to the subsystem under test (supporting hierarchy and key units) are allowed to be executed. Hence test code is inserted into the system to ensure that those plans that do not belong to the subsystem under test are never executed.

In the code augmentation process, the names of the plans that do not belong to the subsystem under test are obtained from the design document of the system. Then test code is inserted to the implemented code of each of these plans, to make the plan never be applicable for its triggering event. In other words, the value of the plan’s context condition is always false. Consequently, the plan is never executed. In JACK, the `context()` method of a plan evaluates the plan’s context condition. Hence, test code is
The original code of the plan:
public plan Plan.One extends Plan {
    ....
    #handles event Trigger trigger.ev;
    context() {
        CC.Expression; //the original expression of the context condition
    }  
    ....
}

After code augmentation:
public plan Plan.One extends Plan {
    ....
    #handles event Trigger trigger.ev;
    context() {
        false && CC.Expression; //the CC value is always false
    }
    ....
}

*Figure 4.24: Test Code for Blocking The Execution of One Plan*

inserted into the `context()` method of each plan, to make it always return false, as the example shown in Figure 4.24.

2. Test Code Inserted

For each testable feature as defined in section 3.2, the associated runtime states and behaviour of the unit under test should be tracked in each test case. Table 4.2 shows the information that needs to be tracked for each testable feature.

Test code is embedded into the system at appropriate locations to track the associated information and report on the information tracked to the test agent. For example, to track the context condition (CC) values of the plan under test, test code is inserted into the method that evaluates the CC value, which is the `context()` method. At runtime such test code logs the CC value and reports on it to the test agent. The details of at which locations and what test code is embedded for tracking runtime information of different types of units are discussed in the following sections.

4.4.1 Plan

When a plan is tested, some of the information needing to be tracked can be directly observed such as whether a plan completes successfully, thus test code can be inserted at appropriate

---

6Features 4.1 and 4.2 are verified by a static test, hence are not presented in the figure.
Chapter 4. Testing Framework

<table>
<thead>
<tr>
<th>Testable Feature</th>
<th>Information Tracked</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1: Is the plan considered based on the event that it is supposed to handle?</td>
<td>whether the plan under test is triggered, and the name of the event that triggers the plan</td>
</tr>
<tr>
<td>1.2: Does the context condition of the plan provide discrimination if present?</td>
<td>the context condition (CC) values of the plan under test</td>
</tr>
<tr>
<td>1.3: Does the plan post the outgoing events specified and only those events?</td>
<td>The posting of outgoing events in the plan under test and their types</td>
</tr>
<tr>
<td>1.4: Does the plan complete?</td>
<td>whether the plan under test completes successfully or not</td>
</tr>
<tr>
<td>2.1: Does a cyclic execution exist at runtime?</td>
<td>The number of cyclic executions</td>
</tr>
<tr>
<td>2.2: Does the cyclic execution path terminate?</td>
<td></td>
</tr>
<tr>
<td>3.1: Is there always an applicable plan for the event?</td>
<td>the CC value of each plan that handles the event under test</td>
</tr>
<tr>
<td>3.2: Is there more than one applicable plan for the event?</td>
<td>which plan is executed in each test case</td>
</tr>
<tr>
<td>3.3: Does a plan that handles the event under test never executes?</td>
<td></td>
</tr>
<tr>
<td>3.4: For each action rule, does the belief post the event specified in the rule when and only when the operation specified succeeds?</td>
<td>success of the operation and the posting of the event</td>
</tr>
</tbody>
</table>

Table 4.2: Information Tracked at Runtime for Testable Features

locations to report on associated values. Other information is not directly observable such as the context condition value. Test code may need to be inserted to track the information required to reason about the relevant issue.

Directly Observable Information

- whether the plan under test is triggered, and the name of the triggering event

To track whether the plan under test is triggered by the event that the plan is supposed to handle (Feature 1.1), the test harness needs to check if the plan under test is selected for evaluating its applicability for the triggering event (in other words, if the context condition is evaluated). Hence test code is inserted at the location where the context condition is evaluated.

In JACK, test code is inserted in the `context()` method of the plan under test, to log the actual triggering event (see Figure 4.25). When the plan is evaluated for its applicability, the `context()` method will be invoked and the “`BIT_log_PlanReceivedMsg()`” method in the test code will be invoked sending a result message (see `Res_Trigger_Msg` in Figure 4.26) that carries the name of the triggering event to the test agent. Then the `PT_Trigger_Handle` plan in the test agent (see Figure 4.26) handles this result message and logs the name of the triggering event.

(October 14, 2011)
In a test case the test agent will not receive a result message if the plan under test is not activated and evaluated for its applicability. If this happens, a time-out mechanism is used to control the termination of the test case and we conclude that the plan under test is not triggered by its expected triggering event in the test case.

If in any test case the plan under test is not triggered by its expected triggering event, a fault of “The plan is not considered based on its triggering event as specified.” will be identified (refer to the fault type FT_TRIGGER in Table 3.1).

- whether a plan completes successfully or not

To verify the completion of a plan (Feature 1.4), test code is inserted at a location where the code is executed after the plan completes or fails to complete. JACK provides two member methods `pass()` and `fail()`: `pass()` is invoked when the plan completes successfully, while `fail()` is invoked if the plan fails to complete. Test code is inserted at the beginning of these two methods to inform the test agent if the plan completes successfully or not (see Figure 4.27), by sending a result message (see `Res_Completion_Msg` in Figure 4.26).

If in any test case, the test agent receives a message indicating that the plan fails to complete, a fault that the plan under test fails to complete will be identified (refer to the fault type FT_COMPLETION in Table 3.1).

**Information Not Directly Observable**

- the context condition (CC) value of the plan under test

To evaluate if the CC of the plan under test is providing discrimination if present (Feature 1.2), the CC value of the plan under test needs to be observed. However, the CC value cannot be directly observed anywhere in the subsystem under test. In general the CC is represented as a logical expression that returns a `true` or `false` value. To detect the value of the CC expression, test code can be inserted into the plan under test to encapsulate the CC expression as a parameter of a method, then in the method the CC value can be obtained, such as the “`BIT_log_CC()`” method shown in Figure 4.28.

However, in many agent development platforms such as JACK, a CC expression can contain unbound (logical) variables that are bound when the CC is evaluated. Consequently a CC containing such variables cannot be directly used as a parameter of a method. Hence, we use an alternative strategy to encapsulate the CC expression as
public plan Plan_Under_Test extends Plan {
    #handles event TriggerType trigger_ev;
    context() {
        /* Test Code - report on the received trigger to the test agent. 
        This method always returns true. */
        BIT_log_PlanReceivedMsg("TriggerType")
        &&
        /**Test Code - End. Below is the original content*/
        true;
    }
}

Figure 4.25: Test Code for Reporting The Trigger of The Plan under Test

Figure 4.26: Test Harness for a Plan (Detailed Structure)
public plan Plan_Under_Test extends Plan {
    ....
    #reasoning method body() { .... }
    ....
    #reasoning method pass() {
        /* Test Code - report on the plan completion to the test agent */
        BIT_log_Completion(true);
        /* Test Code - End */
        ....
    }
    ....
    #reasoning method fail() {
        /* Test Code - report on the plan completion to the test agent */
        BIT_log_Completion(false);
        /* Test Code - End */
        ....
    }
}

Figure 4.27: Test Code for Reporting Plan Completion

shown in Figure 4.29. In the “BIT_log_CC()” method, the CC value is obtained and is carried by a result message that is then sent to the test agent (see Res_CC_Msg in Figure 4.26). Finally the method returns the same value as the CC value.

In each test case, test code reports on the CC value to the test agent. After all test cases are executed, the test agent will check if a fault relevant to Feature 1.2 exists (refer to Table 3.1).

• posting of outgoing events (associated with Feature 1.3)

To track if an outgoing event is posted, two aspects need to be considered:

– if the code for posting the event is invoked or not
– if the event is successfully posted or not

To track the first aspect, test code is inserted immediately before every statement that posts an event in the implemented code of the plan, as shown in Figure 4.30. The test code inserted sends a result message (see Res_Out_Msg in Figure 4.26), which carries the name and the type of the outgoing event to be posted, to the test agent.

In order to check the second aspect regarding whether an outgoing event is successfully posted, those sub-plans that are supposed to handle the event will be monitored to check if these plans are assessed for relevance to the event. If they are we can conclude
The original code of the plan under test:

```java
public plan Plan_Under_Test extends Plan {
    ....
    context() {
        CC_Expression; //the expression of the context condition
    }
    ....
}
```

After code augmentation:

```java
public plan Plan_Under_Test extends Plan {
    ....
    context() {
        /* the original CC expression is encapsulated as a parameter */
        BIT_log_CC(CC_Expression);
    }
    ....
}
```

**Figure 4.28: Test Code for Reporting on The CC Value of The Plan under Test**

Alternative test code for encapsulating the CC:

```java
public plan Plan_Under_Test extends Plan {
    ....
    context() {
        /* the original CC expression is encapsulated as a parameter */
        (CC_Expression)?BIT_log_CC(true):BIT_log_CC(false);
    }
    boolean BIT_log_CC(boolean CC_result) {
        /* the method below sends a message to the test agent to report on the CC value */
        inform_testagent_CC(CC_result);
        return CC_result;
    }
    ....
}
```

**Figure 4.29: Test Code for Reporting on The CC Value of The Plan under Test (Alternative)**
the outgoing event was successfully posted. In JACK, a static method \texttt{relevant()} to a plan is invoked when its triggering event is successfully posted, in order to evaluate the relevance of the plan to the event posted\footnote{This is evaluated before the context condition of the plan is evaluated.}. Hence test code is inserted into the \texttt{relevant()} method of each sub-plan of the plan under test to track what associated outgoing events are posted, as in the example shown in Figure 4.31. The “\texttt{BIT}\_\texttt{log}\_\texttt{Relevance}” method sends a result message (see \texttt{Res\_Relev\_Msg} in Figure 4.26) to the test agent, reporting on the fact that the plan has been evaluated for its relevance to the event it handles. If the test agent receives a \texttt{Res\_Relev\_Msg} message, it will identify that the associated outgoing event has been successfully posted. We do not verify whether the outgoing event activates appropriate plans for execution, as such an aspect has been verified when the event and the associated sub-plans were tested\footnote{According to the testing order that has been determined, all sub-plans and outgoing events of a plan should have been tested before the plan itself is tested, refer to section 4.2.}.

In each test case, if the code for posting an outgoing event is invoked and such an event is successfully posted, the test agent identifies that the plan under test actually posts that event. After the execution of all test cases, the test agent will compare all the actually posted outgoing events against the expected outgoing events as specified in the design documentation. It is not necessary that all the outgoing events specified in the design are posted in every execution of the plan under test (every test case). However, it is expected that all these events are posted across the set of all test cases for that plan. If this does not happen, a fault of a specified outgoing event not posted will be
identified (refer to the fault type `FT_OUTEV_NEVER` in Table 3.1). In addition, if an event posted at runtime is not specified in the design as an outgoing event of the plan under test, a fault of a non-specified event being posted will be identified (refer to the fault type `FT_OUTEV_NOT` in Table 3.1).

### 4.4.2 Plan Cycle

The existence of a cyclic execution is tracked by checking the number of times the cyclic plan under test is executed. When a test case is executed, the execution path of the plan cycle starts from the execution of the cyclic plan under test. A cyclic execution exists if the cyclic plan is executed multiple times. Hence test code is inserted into the code of the cyclic plan under test, to track the number of times the cyclic plan is executed, as shown in Figure 4.32. If the cyclic plan under test is executed more than once, the “`BIT_log_Cycle()`” method will send a result message (see `Res_Cycle_Msg` in Figure 4.33) to the test agent informing the existence of a cyclic execution. If the test agent never receives a `Res_Cycle_Msg` message in a test case, a fault of absence of cyclic executions is identified (refer to the fault type `FT_CY_EXISTENCE` in Table 3.2).

The infinite execution of the cycle path cannot be directly tracked. Instead, we introduce a pre-defined maximum limit, which can be specified by the user, for the number of iterations that occur in a cyclic execution path (refer to section 3.2.2). We check if the number of times any plan in the cyclic structure is executed exceeds the maximum limit. Every plan in the cyclic structure needs to be checked. This is because the plan cycle under test may be a combination of multiple cycles and the infinite execution may be in a sub-cycle that does not contain the current cyclic plan under test. For the example in Figure 4.34, `P6` is the current cyclic plan under test. However, after `P3` is triggered by the event `e3` from `P6`, the cyclic
execution may follow the path “P3 → P1 → P2 → P3 . . .” and may never stop.

Test code is inserted into every plan in the cyclic structure to check if the number of times the plan executes exceeds the maximum limit, as the code shown in Figure 4.35. If this happens, the “BIT_log_NoStop(“) method sends a result message (see Res_Cycle_NoStop_Msg in Figure 4.33) to the test agent, which then identifies a fault of “The iterations of the cyclic execution path exceeds the pre-defined maximum limit.” (refer to the fault type FT_CY_NO_STOP in Table 3.2).

### 4.4.3 Event

An event test harness checks overlap and coverage of the event under test, and whether each plan that handles the event executes in some situation. In order to verify whether overlap or incomplete coverage occurs when an event is tested (associated with Feature 3.1 and 3.2), in each event test case the test harness should track the context condition value of each plan that handles the event under test. Test code is inserted into each plan that handles the event under test, following the same strategy as that for detecting the context condition value when a plan is tested (refer to section 4.4.1). The internal structure of the test agent is shown in Figure 4.36.

---

```
public plan Plan_Under_Test extends Plan {
    ....
    public static int numOfExec = 0; // the number of times the plan is executed

    /* Test Code - check cyclic execution */
    reasoning method body() {
        /* Test Code - check cyclic execution */
        numOfExec++;
        if(numOfExec > 1) {
            BIT_log_Cycle(numOfExec);
        }

        // If the number of cycles exceeds the maximum limit, the execution will be terminated.
        if(numOfExec > getCycleMaxLimit()) {
            BIT_log_NoStop();
            terminateExecution();
        }
    }

    /* Test Code - End. Below is the original code of the plan body */
    ....
    }

Figure 4.32: Test Code for Reporting The Cyclic Execution
```
At the end of a test case, the test agent identifies overlap or incomplete coverage according to the number of applicable plans. The test agent also accesses the design specification to check if overlap and incomplete coverage are allowed for the event under test and may identify event-specific faults of these two situations according to the specification (refer to Table 3.3).

To verify “Does a plan that handles the event under test never execute?” (Feature 3.3), the test agent needs to know which plan executes in each test case. Hence test code is inserted into each plan that handles the event under test at the beginning of the plan’s body (see Figure 4.37), informing the test agent when the plan is executed. In addition, other code in the plan’s body will not be executed because the test agent only needs to know if the plan is
executed or not. If there is one plan that is never executed in any test case, a fault associated with Feature 3.3 will be identified (refer to Table 3.3).

4.4.4 Belief

A belief test harness verifies each action rule of the belief under test. In a test case, the belief test harness first changes the belief according to the operation specified in the associated action rule, then tracks the success of that operation and posting of events from that belief.

In each test case, the test-driver plan creates an instance of the belief under test, then performs the operation specified in the relevant action rule: inserts, deletes or updates a record in the belief under test and logs this record for later checking. For a delete or update operation, a record is inserted into the belief for the purpose of being deleted or updated later. The value of the record inserted is from the test inputs of the test case. The generation of test inputs is discussed in chapter 5.

After the operation has been performed, the test-driver plan accesses the belief to check if the operation has succeeded or not, and reports on the result to the test agent by sending a result message (see Res_Opera_Msg in Figure 4.38).

After the operation is performed by the test-driver plan, the event that is specified in the relevant action rule may or may not be posted. The test code that has been inserted into the belief logs the posting of an event and informs the test agent of the event posting by sending
public plan Plan_Handle_Event extends Plan {
    
    // reasoning method body() {
    /* Test Code - report on the plan execution to the test agent*/
    BIT_log_Plan_Exec(BIT_getCurrentPlanName());
    return true; //the execution is terminated.
    /** Test Code - End. Below is the original code of the plan body*/
    
    }  
    ....
}

Figure 4.37: Test Code for Reporting on The Execution of A Plan

another result message (see the Res_Event_Msg message in Figure 4.38). The code insertion follows the same strategy as that for tracking the outgoing events of a plan under test (refer to section 4.4.1): the test code is inserted immediately before every statement which posts an event in the belief.

The test agent will identify the faults relevant to whether the belief posts an event following each action rule according to two factors: if the operation succeeds or not and if the belief posts out the event as specified or not (see Table 3.2.4). The success of the operation will be checked by the test-driver plan after it performs the operation, as mentioned above. The test agent will identify that the belief does not post out the event as specified in two situations: no Res_Event_Msg message is received or one or more Res_Event_Msg messages
are received but none of them carries the name of a posted event that is the same as that specified in the design.

4.4.5 Implementation of Mock Agents

When a plan is executed during its testing process, it may interact with external agents in the same system. These interactions may be essential to the internal logic of the plan, as the plan may wait for a reply after sending a message to an external agent, to continue its execution. For example, in an online bookstore system, the Query Book by Name plan sends a Book Query message to the Stock Manager agent to check if a book queried by a user is still in stock (see Figure 4.39). If the plan does not receive any reply from the agent in a given time period, it will fail to give the user the result.

It may be the case that the agent receiving the message has not yet been tested or implemented. Also, interactions between the agent under test and other agents are not within the scope of unit testing and therefore are not verified in the testing framework.

A mock agent is introduced in the testing framework as a simple proxy to simulate an external agent that interacts with the plan under test. Such an external agent is called an interactee agent. During the code augmentation process for a plan, a mock agent is generated to replace each interactee agent. A mock agent simulates the message-reply logic of its associated interactee agent. When the plan under test is executed during its testing

\[\text{mock agent} \rightarrow \text{interactee agent}\]

\footnote{The user can also specify to use the original code of a particular interactee agent if the code of the agent has already been tested.}
process, any message sent by the subsystem under test to an interactee agent will be received by the respective mock agent. When the mock agent receives a message, it will have one of two possible responses which is dependent on if the plan under test is waiting for a response or not:

- If the plan under test is waiting for a reply message, according to the associated protocol specified in the design and the implemented code of the plan, the mock agent will send a reply message to the agent under test. If the reply message contains some variables, the values of the variables will be random. This is because the values of these properties may be dependent on the internal logic of an interactee agent, which usually cannot be precisely represented in the design specification. Hence it is difficult to simulate reply values in the implementation level. Therefore mock agents only simulate the message-reply operations of their associated interactee agents without consideration of the values replied.

- If the plan under test does not wait for a reply message, the mock agent will log the message received and do nothing else.
Mock agents are not necessary for an event or a belief. As mentioned above, testing an event does not require the complete execution of the plans that handle the event, so communications with external agents do not occur. In testing a belief, the execution of a plan is unnecessary, therefore mock agents are also not required in this case.

During the code augmentation process that implements a test harness for the plan under test, the code of each interactee agent will be replaced by the code of an associated mock agent that is automatically generated. Generating the code of mock agents consists of the following steps. We use the example in Figure 4.39 to explain these steps:

- **Step 1: Identify interactee agents of the plan**
  
  All messages sent from the subsystem under test to external agents are extracted and the relevant interactee agents which receive these messages are identified. In the example the plan under test *Query Book by Name* (see Figure 4.39) has two messages: *Book Query* and *Book Order*, which are both sent to the *Stock Manager* agent, which therefore is an interactee agent.

- **Step 2: Generate the code of mock agents**
  
  For each interactee agent, the testing tool generates code for the mock agent that replaces interactee agent. The code generation follows the rules below:

  1. The mock agent shares the same name as the associated interactee agent. In the example, there is one mock agent that replaces and takes the name of the *Stock Manager* agent.
  2. If this interactee agent has been implemented, it may have some user-defined constructors. The associated mock agent will have the same constructors.

- **Step 3: Generate the code of the plans that handle outgoing messages**
  
  For each outgoing message from the plan under test and received by the interactee agent, one plan that handles this message is declared in the associated mock agent and the code of the plan is generated.

  The newly generated plan may send back a reply message to the agent under test if the agent awaits a reply. In this case the reply message is extracted by parsing the code of the plan under test and code is inserted into the newly generated plan to send the reply to the agent.
CHAPTER 4. TESTING FRAMEWORK

//definition of the mock agent
public agent StockManager extends MockAgent {
    uses plan HandleBookQuery;
    uses plan HandleBookOrder;
    ... // other constructors etc.
}

//the plan that handles the “BookQuery” message
public HandleBookQuery extends Plan {
    #handles MessageEvent BookQuery inMsg;
    relevant( ) { //the condition to only accept
        // the message from the agent-under-test
    }
    body( ) {
        log(inMsg); //log the message received;
        //create a replying message with a random value
        BookQueryResponse reply = createRandomValueOfBookQueryResponse();
        if(reply !=null){
            send(agent-under-test, reply);
        }
    }
} //end of plan

//the plan that handles the “BookOrder” message
public HandleBookOrder extends Plan{
    #handles MessageEvent BookOrder inMsg;
    relevant( ) { //the condition to only accept
        // the message from the agent-under-test
    }
    body( ) {
        log(inMsg); //log the message received
    }
} // end of plan

Figure 4.40: Sample Mock Agent Code: StockManager

In the example of Figure 4.39, two plans are declared that respectively handle the messages Book Query and Book Order: the Handle Book Query plan that handles the Book Query message and the Handle Book Order plan that handles the Book Order message (see the code in Figure 4.40). The Handle Book Query plan replies with the Book Query Response message according to the Book Finding protocol specified in the design document (see Figure 4.39) and the source code of the plan under test Query Book by Name (see the code in Figure 4.41). The Handle Book Order plan only logs the received message and does not reply, as no reply is extracted.

• Step 4: Replace the code of an interactee agent
CHAPTER 4. TESTING FRAMEWORK

After the generation process above, the code of each mock agent is embedded into the test implementation of the system to replace the code of the associated interactee agent.

As mentioned above, a reply message generated by a mock agent carries variable values that are randomly generated, because a mock agent does not implement the logic of its associated interactee agent. As a result, the plan under test may not work properly if the plan’s logic depends on a reply and spurious faults may be detected. If this happens, the user should investigate the fault detected to discover the exact reason that lead to the fault.

Implementation of mock agents is a supplement to the unit testing framework. This supplement partially addresses the issue that interactions between agents are not within the scope of unit testing but may be necessary to the successful execution of the plan under test, thus providing more realistic testing.

4.5 Test Report

The results of testing an agent system (or partial system) are summarised in a test report that is automatically generated by the testing tool. The test report is in HTML format and describes the test results in three levels:
CHAPTER 4. TESTING FRAMEWORK

Figure 4.42: Initial Page of A Test Report

- The System Level:

In the first page of the test report, an overview of the system under test is presented (see Figure 4.42), including the names of the design specification and the implemented code of the system. These names are hyperlinked to their locations in the file system of the computer. In the list of agents, the outline of unit tests for each agent is also shown, including the agent name that is hyperlinked to the associated agent report, the number of the units having been tested within the agent and the number of faults detected.

113 (October 14, 2011)
• The Agent Level:

An agent report describes the overview of the agent and the outline of testing each unit (see Figure 4.43). In the agent report the units that have been tested are categorised into three groups according to the faults detected: the units that have exceptions or errors (level-1 or level-2 faults), the units that only have warnings (level-3 faults) and the units without faults. For each unit, the number of test cases generated, the number of faults detected and a short description for each fault is given. The name of each unit is hyperlinked to the associated unit test report.

![Agent Test Report](image)

**Figure 4.43: Example of An Agent Test Report**

• The Unit Level:

A unit test report presents the details of a unit testing process and consists of four parts:

- the overview of the units under test describing the key units and the number
CHAPTER 4. TESTING FRAMEWORK

Figure 4.44: Example of A Unit Test Report (Overview and Test Inputs)

of test cases generated

– the test inputs, including the details of input variables and combinations of input variable values as inputs for test cases.

– the test summary, which consists of the result of verifying each testable feature for the different types of units. For example, in the report of a plan, the summaries are associated with the plan’s four testable features, which are relevant to the plan’s triggering event, context condition, outgoing events and completion (refer to section 3.2.1).

– the log of all test cases, in which the details of each test case are given.

An example of a unit test report is shown in Figure 4.44 and Figure 4.45.
A test report facilitates the developer to understand errors in the system, as these errors have been indicated by the faults described in the report. The report also facilitates the developer to debug the system and fix up errors, because the details of the test cases that are associated with each fault, which are also described in the report, reveal the situation that errors occur.

A more extensive example of a test report can be seen in Appendix E.

Summary

In this chapter, we presented the automated testing framework developed for unit testing an agent system. The testing framework identifies the units to be tested within an agent system and determines the order in which these units are to be tested. Then for each unit, a test harness program is automatically implemented which tests the unit to verify the relevant
testable features described in the fault model (refer to section 3.2). The test results of all units are summarised in a test report in HTML format.
Chapter 5

Test Input Generation

This chapter examines the generation of inputs for test cases. The inputs of each test case are composed of the values of input variables, which are those variables that are not the local variables\(^1\) of the unit under test and its descendent units, but are accessed by the unit under test or its descendent units. For example, the input variables for testing a plan include the public variables of the triggering event and the global variables that are referenced in the plan’s context condition or body and the belief variables accessed by the plan, as well as such variables accessed by the descendent plans of the plan under test. The developer is required to declare the input variables for each unit to be tested in the design descriptor of the unit. Details regarding variable declaration are discussed in section 5.1.

In the testing framework, inputs of the test cases for a unit are automatically generated based on types and value ranges specified in the declaration of input variables. Ideally, test inputs generated should cover the entire spectrum of situations that may be encountered by the system under test. However, it is generally impossible to cover all possible situations. For example, the domain values of a variable are usually continuous and there are an infinite number of possible values. In addition, the inputs of a test case are usually a combination of values of input variables and the number of possible combinations exponentially increases with the increase in the number of input variables and the number of possible values of each variable. If a unit under test has ten input variables for instance, each of which has five possible values, there will be millions of possible value combinations of variables (\(5^{10}\)). Hence an appropriate mechanism is required for generating a finite set of test case inputs to cover the spectrum as sufficiently as possible.

\(^1\)A local variable is one that is declared inside the unit under test.
CHAPTER 5. TEST INPUT GENERATION

An approach has been used in the testing framework to rationally choose finite samples of values for each input variable, based on the value domain information specified in the input variable declaration of the unit under test. We then combine these variable values in a way to obtain a manageable number of test cases. The details of these approaches are discussed in section 5.2.

In addition to the automatically generated test cases, the developer is also allowed to manually add extra test cases by specifying particular values of each input variable, especially when the developer thinks that some particular situations that should be checked may not be covered by the automated test cases. Section 5.2.1 discusses how the developer adds manual test cases.

The generation of test case inputs also considers relationships between values of different input variables. There are two different situations relevant to relationships between variables. One is that a constraint exists between values of two input variables. For example, a Buy A Book plan has \( \minPrice \) and \( \maxPrice \) amongst its input variables, indicating the lower bound and the upper bound of the price of a book. The values of these two variables maintain the constraint that \( \minPrice \leq \maxPrice \). Comparisons like this indicate valid and invalid interactions between variables, because it is valid if \( \minPrice \) is no more than \( \maxPrice \), else, it is invalid. Another situation is where a plan may execute different paths according to different comparative relationships between values of variables. For instance, a Response On Stock plan in a Stock Management agent checks the number of books in stock and gives responses according to different numbers. The plan has an input variable \( \text{numInStock} \) that indicates the actual number of books in stock and another input variable \( \text{limit} \) that indicates the minimum number of books that should be in stock. If \( \text{numInStock} < \text{limit} \) then the plan will activate a subtask to buy new books; else, the plan will make a log and do nothing more. These two situations are both taken into account when input variables are specified and the inputs of test cases are generated as discussed in section 5.3.

In the runtime environment, values of input variables are assigned to the implementation of the associated variables as the inputs to the test. As a part of the automated testing process, the assignment of input variable values is automatically carried out at the beginning of a unit testing process. Details of how variable values are assigned are discussed in section 5.4.
5.1 Input Variable Declaration

Before an agent system is unit tested, the designer/tester can declare the variables accessed by each unit in the unit’s design descriptor. When a unit is tested, its variables and those of other key units will be extracted by the test harness for the generation of test inputs.

Input variables are declared according to which variables are referenced within the unit to be tested and which variables affect the runtime environment under which the unit is executed. There are different input variables for various types of units (plans, events and beliefs). Input variables for a plan are the variables that are referenced in the context condition and the body of the plan to be tested and those of its descendnet plans. All input variables are declared as part of the design descriptors of the plan to be tested or its descendant plans in a plan’s context condition entity (if the variable is referenced in the plan’s context condition), or in a plan’s procedure entity (if the variable is referenced in the plan’s body), as shown in Figure 5.1. When a plan is tested, input variables of the plan are extracted from the design descriptors of the plan and its descendant plans. Test inputs are then generated for these variables.

In testing an event, we are interested in the coverage and overlap of the event (refer to section 3.2.3). Therefore the variables in the context conditions of the plans that handle the event are important. These variables are declared in the design descriptors of the relevant plans and extracted as the input variables of the event.

When a belief is tested action rules of the belief are verified as discussed in section 4.3.1. The action specified in a rule may or may not be activated at runtime in different situations depending on the values of belief fields or global variables. Therefore input variables of a belief are those belief fields or global variables that may affect the activation of an action rule and such variables should be declared in the design descriptor of the belief.

The Format of Variable Declarations
A declaration of input variables consists of four parts as presented in the following. The Backus-Naur Form (BNF) of a variable declaration is presented in appendix B.

1. Definition of enumerated types:

This part is declared if some variables are of user-defined enumerated types. The

2 A unit may have no input variables, such as a plan that does not access any belief, any global variable and any event variable.

3 Key units are the unit under test and other relevant units whose runtime behaviours or states are tracked and logged for testing purposes (refer to section 4.3.3).
CHAPTER 5. TEST INPUT GENERATION

(a) Variables in A Plan’s Context Entity

(b) Variables in A Plan’s Procedure Entity

Figure 5.1: Declaration of Input Variables for A Plan

definition of each enumerated type contains the name and the value domains of the
type, such as [EnumType, SupplierType, {“Amazon”, “Angus&Robertson”, “Powells”,
“Dymocks”}].

2. Definition of input variables:
This part consists of a list of input variables. Each input variable is specified as a tuple
[scope, name, type, domain-info], such as [event-var, minPrice, float, {>= 1.0}].
CHAPTER 5. TEST INPUT GENERATION

- **scope** - is one of *agent-var, event-var, belief-var, or system-var*, which denotes the scope of the variable. For example, a plan that handles a particular event may rely on variables contained in the event (*event-var*), variables of the agent that contains this plan (*agent-var*), variables that are fields of a belief (*belief-var*), or global variables in the system (*system-var*).

- **name** - is the string identifier of the variable, for example, *price*, or *book.bookName* for (nested) structures, such as a property of an object.

If the variable is a field of a belief (*belief-var*), then the **name** is a tuple (*belief-name, belief-var-name, field-name*), where *belief-name* is the name of the belief, *belief-var-name* is the name of the belief in the implemented code of the system and *field-name* is the name of the belief field. For example (*StockDB, stock_data, numberInStock*).

- **type** - specifies the type of the variable. We allow for the base types: *integer, long, float, double, boolean, string, array*, enumerated types as discussed above and types of user defined classes.

- **domain-info** - for all types other than an array, this is the domain of the variable values, consisting of a list of expressions that define value ranges of the associated variable, such as “>0” or “>=0” for a numeric variable (integer and float) and “!=null” for a string variable.

For an array variable, this is a tuple (*size, element-type, domain-info*) that respectively specifies the size of the array, the type of the elements and the domain of the values of each element. For example (*5, float, {>0.0, <50}*).

For an object variable whose type is a user defined class, its value range is a list of instances of the class or its descendant classes. An instance can be declared with arguments of a constructor of the relevant class. In the following example, the value range of the *vehicleVar* variable includes an instance of the *Vehicle* type and an instance of the *Car* type, which is a subclass of *Vehicle*:

- [*agent-var, vehicleVar, Vehicle, {== “Vehicle()”, == “Car(‘myCar’)”}*]

3. Definition of comparative statements:

This part describes comparative relationships between variable values. There are two

\[\text{in terms of Object Oriented Programming.}\]
different situations of relationships between variable values, as mentioned in the introduction of this chapter. Two kinds of comparative statements are defined to respectively describe these two situations.

A comparative statement of the first kind describes valid and invalid relationships between the values of two variables. The statement is specified as a tuple ("compare", comparative-expression), where "compare" is the keyword and comparative-expression specifies the valid comparative relationship between two variables. For example, the statement [compare, minPrice ≤ maxPrice] specifies that minPrice ≤ maxPrice is valid and implicitly indicates that minPrice > maxPrice is invalid. The comparative operation between two variables is one of <, >, !=, ==, >= and <=. The expression in each side of the comparative operation can also be a mathematical expression of an input variable, such as [compare, minPrice < 10*minPrice/2] or [compare, minPrice < maxPrice/2-1.0].

The second kind of comparative statement is specified for the situation that the unit under test may execute in different paths according to different relationships between variable values, such as the example of minPrice, maxPrice and queryPrice mentioned in the introduction. A statement of this kind is a tuple ("compare", variable-list), where variable-list lists the input variables that may affect execution paths of the unit under test, such as [compare, minPrice, maxPrice, queryPrice]. The statement indicates that the comparison between each pair of variables in the list should be taken into account when test inputs are generated.

Each variable specified in a comparative statement should be an input variable that has already been declared.

4. A textual description, which is the description of the unit specified by the user.

Each part of a declaration is not mandatory and the developer specifies some or all of them as necessary. Also, a declaration for the unit under test is not mandatory as the unit may have no input variables. In such cases the testing harness of the unit generates one test case, which does not carry any variable values for the unit under test.

Below is an example of a variable declaration:

\[\text{[EnumType, SupplierType, \{"Amazon", "Angus&Robertson", "Powells", "Dymocks"\}]}\]

\(^5\)The operators used here follow the Java programming standard.
5.2 Test Input Generation

In a unit testing process, the next step after the extraction of the input variables is the generation of the values for these variables as inputs of test cases. The standard notions of Equivalence Class Partitioning and Boundary Value Analysis are used to generate a more limited set of values for each relevant variable. In addition, comparative statements are also taken into account in the generation of test cases and the details regarding this are discussed in the next section.

Equivalence Class Partitioning is the principle that the input domain of a variable can be partitioned into a finite number of Equivalence Classes (EC), each of which is a range of values such that if any value in that range is processed correctly (or incorrectly) then it can be assumed that all other values in the range will be processed correctly (or incorrectly) [Myers et al., 2004, page 54]. This principle has been widely used for identifying test inputs [Jorgensen, 2002, page 99; Burnstein, 2002, page 67; Copeland, 2004, page 39].

Boundary Values mark the (non-infinity) ends of an EC and are often values that cause errors [Burnstein, 2002, page 72]. Some approaches to testing use only valid ECs that indicate valid value ranges [Jorgensen, 2002, page 98], while others improve robustness by also using invalid ECs which indicate invalid value ranges [Jorgensen, 2002, page 99; Burnstein, 2002, page 67; Copeland, 2004, page 39]. Based on these approaches, in our testing framework, the user can select from three different levels for choice of values for variables:
CHAPTER 5. TEST INPUT GENERATION

1. *Minimal level* is restricted to valid values and uses boundary values for the valid ECs.

2. *Normal level* uses both valid and invalid ECs, selecting boundary values and values close to the boundary, for each EC.

3. *Comprehensive level* consists of the normal level values together with the midrange value of each EC if none of the EC’s boundaries is \(-\infty\) or \(\infty\), else, it is equivalent to the normal level.

To illustrate the above let us consider the following specification:

\[
\begin{align*}
&[\text{agent-var}, \text{bookID}, \text{int}, \{>0\}] \\
&[\text{agent-var}, \text{stock}, \text{int}, \{\geq 0, \leq 200\}] \\
&[\text{agent-var}, \text{price}, \text{float}, \{>0.0, \leq 90.0\}]
\end{align*}
\]

Equivalence classes for each input variable are partitioned by its specified value ranges. Table 5.1 gives their ECs and the choice of values from the three levels.

<table>
<thead>
<tr>
<th>name</th>
<th>index</th>
<th>value range</th>
<th>valid</th>
<th>minimal</th>
<th>normal</th>
<th>comprehensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>bookID</td>
<td>EC-1</td>
<td>&lt;0</td>
<td>no</td>
<td>N/A</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>EC-2</td>
<td>(\geq 0)</td>
<td>yes</td>
<td>0</td>
<td>0, 1</td>
<td>0, 1</td>
</tr>
<tr>
<td>stock</td>
<td>EC-1</td>
<td>&lt;0</td>
<td>no</td>
<td>N/A</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>EC-2</td>
<td>(0 \leq \leq 200)</td>
<td>yes</td>
<td>0, 200</td>
<td>0, 1, 199, 200</td>
<td>0, 1, 100, 199, 200</td>
</tr>
<tr>
<td></td>
<td>EC-3</td>
<td>&gt;200</td>
<td>no</td>
<td>N/A</td>
<td>201</td>
<td>201</td>
</tr>
<tr>
<td>price</td>
<td>EC-1</td>
<td>(\leq 0.0)</td>
<td>no</td>
<td>0.0</td>
<td>-0.1, 0.0</td>
<td>-0.1, 0.0</td>
</tr>
<tr>
<td></td>
<td>EC-2</td>
<td>&gt;0.0 (\leq 90.0)</td>
<td>yes</td>
<td>90</td>
<td>0.1, 89.9, 90</td>
<td>0.1, 45.0, 89.9, 90.0</td>
</tr>
<tr>
<td></td>
<td>EC-3</td>
<td>&gt;90.0</td>
<td>no</td>
<td>90.1</td>
<td>90.1</td>
<td>90.1</td>
</tr>
</tbody>
</table>

*Table 5.1: Example of Equivalence Classes*

Having chosen test values for each relevant variable, these values must be combined in various ways to produce the inputs of test cases. Some approaches to testing simply ensure that for each variable, there is at least one test case with each of the values chosen [Jorgensen, 2002, page 98; Burnstein, 2002, page 71; Myers et al., 2004, page 55]. A more thorough approach recognises that many errors are a result of interactions between variable values and takes the cross product of all values for each variable. The problem is that the latter quickly gives a very large number of test cases. The first approach caps the number of test cases at the largest number of values for any variable. The second is the cartesian product of the number of values for each variable, which quickly explodes. For example if there are five variables, each with five values, it gives over 3,000 test cases.
A commonly used approach to reduce the number of combinations is combinatorial design [Cohen et al., 1997]. This approach generates a new set of value combinations that cover all n-wise (n≥2) interactions among the test parameters and their values in order to reduce the size of the input data set. The CTS (Combinatorial Testing Service) software library of Hartman and Raskin⁶ is used that implements this approach. However this reduction is applied only to test cases involving invalid values. All combinations of valid values are used, on the assumption that, for agent systems, it is interactions between valid variable values which will cause different parts of the code to be activated (most commonly different plans chosen) and that covering all such interactions is necessary in order to adequately evaluate whether the system is behaving correctly.

Test input of a test case is a combination of variable values. A value combination is valid if all values in the combination are valid with respect to the value ranges of the associated variable. An invalid value combination is one that contains at least one invalid value with respect to the value ranges of the associated variable. In the testing framework, the user can choose appropriate combinations of variable values to obtain different levels of thoroughness in testing. The following options are provided for the choice of value combinations, referring to Jorgensen’s work [Jorgensen, 2002, page 97]:

1. Basic level takes the cartesian product of valid values and then adds one additional case for each invalid value, based on the assumption that invalid values do not require such rigorous testing as valid values.

2. Extended level supplements the results of all invalid value combinations not in the basic set obtained using the 2-wise (or pairwise) reduction of combinatorial design.

In addition, in both levels above, the size of the cartesian product of valid values still may be extremely large if there are too many valid values of each input variable. To cap the size of test cases, the user is allowed to specify a threshold as the maximum limit of the number of test cases generated. If the threshold is exceeded, the pairwise reduction is applied to the cartesian product of valid values when value combinations are chosen.

In the example of Table 5.1, using the values chosen from “Normal” level, the cartesian product of values would give 108 (3×6×6) test cases. Using the “basic” approach above there are 30 test cases (24 (2×4×3) valid and 6 (1+2+3) invalid), as shown in Table 5.2. The “extended” approach gives 52 test cases (24 valid and 28 invalid), as shown in Table 5.3.

The approaches of “basic” and “extended” levels have more significant effects on variables with more values. For example, if there are 4 variables, each of which has 4 valid values and 6 invalid values, the cartesian product of values would give 10,000 test cases. Using the “basic” approach above there are 280 test cases (256 valid and 24 invalid). The “extended” approach gives 366 test cases (256 valid and 110 invalid).

As can be seen, substantial numbers of test cases can be generated, although these can also be controlled if one wishes to do so.
5.2.1 Manual Test Case Input

In addition to the test cases that are auto-generated, the user may wish to specify additional test cases using domain and design knowledge. The testing framework accommodates this by means of a Test Case Input window for a given unit under test.

![Image of Test Case Input windows]

**Figure 5.2: Windows of Manual Test Case Inputs**

Figure 5.2 shows two examples of the Test Case Input window. The example at the left is for a plan that has three variables, *BookID*, *NumberOrdered* and *Urgent*. The user can edit the values in the text box next to the variables and add a new test case using the *Add* button and delete it using the *Remove* button if necessary. The example at the right is similar. Note that for enumerated data types or the boolean type a drop down list is provided allowing the user to select a value. *Urgent* in the left example is of the boolean type and *Supplier* in the right example is an enumerated type. When the user completes editing the test cases, the *Save* button is used to save the information to a file which will be used by the testing framework each time this unit is to be tested, provided that the variable specifications have not changed.
5.3 Comparison between Variables

For a thorough test, the test cases generated should also check comparative relationships indicated by comparative statements between input variables. A comparative statement indicates the validity of comparative relationships between the associated variables. For example, a *Buy A Book* plan has two input variables *minPrice* and *maxPrice* with the constraint that "*minPrice* ≤ *maxPrice*" (the example mentioned in the beginning of this chapter). The specification of the input variables for the plan is shown in Figure 5.3 ⁷. From the constraint of these two variables three basic comparative relationships can be derived, one invalid and the other two valid (see Figure 5.4 ⁸). All comparative relationships derived from test input specification should be taken into account in the generation of test cases. Further, the validity of a value combination must be adjusted with the consideration of comparative relationships. If a value combination with valid variable values matches an invalid relationship, the combination will be invalid. In the example of Figure 5.3 a value combination that contains the values (*minPrice*=20.0 and *maxPrice*=0.0) is invalid as the combination match an invalid comparative relationship (in Figure 5.4).

| ;**; | [event-var, minPrice, float, {>=20.0}] |
| ;**; | [event-var, maxPrice, float, {>=0.0, <=50.0}] |
| ;**; | [agent-var, numberToBuy, int, {>=1}] |
| ;**; | [event-var, varX, int, {>=1}] |
| ;**; | [event-var, varY, int, {>=2}] |
| ;**; | [event-var, varZ, int, {>=1}] |
| ;**; | [compare, minPrice <= maxPrice] |
| ;**; | [compare, varX, varY, varZ] |

Figure 5.3: Comparative Statements

For a statement that indicates the validity of value interactions, valid and invalid relationships between variables are derived from the statement, such as the example of *minPrice* and *maxPrice* mentioned in Figure 5.4. For a comparative statement that indicates relationships between multiple variables, comparative relationships are derived taking into account comparisons between each two variables. From each two variables that are specified in a

---

⁷Other variables are introduced to facilitate the following discussion.

⁸Only basic comparative relationships that are "<", ">" and "==" are considered. The other comparative relationships such as "!=" and "<=" are combinations of basic comparative relationships.
CHAPTER 5. TEST INPUT GENERATION

1. \( \text{minPrice} > \text{maxPrice} \), which is an invalid relationship
2. \( \text{minPrice} < \text{maxPrice} \), which is a valid relationship
3. \( \text{minPrice} == \text{maxPrice} \), which is a valid relationship

**Figure 5.4: Comparative Relationships of The First Kind of Comparative Statements**

comparative statement, three basic comparative relationships are derived, which are all considered as valid because such a comparative statement does not indicate the validity of value interactions between variables. In the example of \([\text{compare}, \text{varX}, \text{varY}, \text{varZ}]\) in Figure 5.3, nine valid comparative relationships are derived from three pairs of variables: \{\text{varX, varY}\}, \{\text{varY, varZ}\} and \{\text{varZ, varX}\} (see Figure 5.5).

1. \( \text{varX} > \text{varY} \)
2. \( \text{varX} < \text{varY} \)
3. \( \text{varX} == \text{varY} \)
4. \( \text{varY} > \text{varZ} \)
5. \( \text{varY} < \text{varZ} \)
6. \( \text{varY} == \text{varZ} \)
7. \( \text{varZ} > \text{varX} \)
8. \( \text{varZ} < \text{varX} \)
9. \( \text{varZ} == \text{varX} \)

**Figure 5.5: Comparative Relationships of The Second Kind of Comparative Statements**

**Test Case Generation with Comparisons**

A comparative relationship derived from a comparative statement indicates a valid or invalid interaction of variable values and therefore should be covered by the test cases generated. In general, a comparative relationship describes a comparison between valid variable values. However, the situation of invalid variable values is also considered when test cases are generated in order to perform a thorough test. Therefore for a comparative relationship, there should be three value combinations that satisfy the comparative relationship as follows:

1. a valid value combination
2. an invalid value combination in which the values of the relevant variables of the c-relationship are all valid.
3. an invalid value combination that contains invalid values of some of the relevant variables of the c-relationship

In order to cover comparative relationships, the set of value combinations generated as discussed in section 5.2 is adjusted by adding new value combinations if necessary. The adjustment process contains four steps, which are illustrated as follows using the example in Figure 5.6, which is a sub-set of Figure 5.3.9:

- **Step 1**: A value combination set is generated for input variables, following the approach as discussed in section 5.2 (see the results in Table 5.4).

- **Step 2**: Each comparative statement is analysed to derive the relevant comparative relationships as shown in Figure 5.7.10.

- **Step 3**: All value combinations (the first 16 items in Table 5.4) and all comparative relationships are checked in order to adjust the validity of value combinations. If a value combination is composed of valid variables but matches an invalid comparative relationship, the combination will be set as invalid. The adjustment results of Table 5.4 are shown as the first 35 items in Table 5.5, with items 9 to 16 adjusted.

- **Step 4**: Then, all valid value combinations (the first 8 items in Table 5.5) and all comparative relationships are checked in order to find out the comparative relationships

---

9 A simpler example is used for illustration since the example in Figure 5.3 will generate too many value combinations.

10 Identical to Figure 4, show again for easy reference.
that are not covered by any valid value combination. In the example, two relationships \( \text{minPrice} == \text{maxPrice} \) and \( \text{minPrice} > \text{maxPrice} \) are not covered. Two new value combinations are generated with valid values covering these uncovered comparative relationship, as the 36-th and 37-th items in Table 5.5. The values in each new combination are randomly created, while maintaining the constraint that the values must satisfy the associated comparative relationship. The 36-th item is valid as it matches a valid comparative relationship (\( \text{minPrice} == \text{maxPrice} \)) and the 37-th item is invalid as it matches an invalid relationship (\( \text{minPrice} > \text{maxPrice} \)).

- **Step 5:** Next all invalid value combinations in which values of \( \text{maxPrice} \) and \( \text{minPrice} \) are all valid are analysed (Items 9 to Item 22 in Table 5.5), together with all comparative relationships. There is no combination that satisfies the relationship \( \text{minPrice} == \text{maxPrice} \). Hence one new invalid value combination is generated to cover this relationship, as the 38-th item in Table 5.5. In the new added combination the values of \( \text{maxPrice} \) and \( \text{minPrice} \) are both valid.

\(^{11}\)The combination is invalid because the value of \( \text{numberToBuy} \) is invalid.
CHAPTER 5. TEST INPUT GENERATION

<table>
<thead>
<tr>
<th>index</th>
<th>minPrice</th>
<th>maxPrice</th>
<th>numberToBuy</th>
<th>valid</th>
<th>compare “minPrice” and “maxPrice”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.0</td>
<td>50.0</td>
<td>1</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>2</td>
<td>20.0</td>
<td>50.0</td>
<td>2</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>3</td>
<td>20.0</td>
<td>49.9</td>
<td>1</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>4</td>
<td>20.0</td>
<td>49.9</td>
<td>2</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>5</td>
<td>20.1</td>
<td>50.0</td>
<td>1</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>6</td>
<td>20.1</td>
<td>50.0</td>
<td>2</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>7</td>
<td>20.1</td>
<td>49.9</td>
<td>1</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>8</td>
<td>20.1</td>
<td>49.9</td>
<td>2</td>
<td>yes</td>
<td>&lt;</td>
</tr>
<tr>
<td>9</td>
<td>20.0</td>
<td>0.0</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>10</td>
<td>20.0</td>
<td>0.0</td>
<td>2</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>11</td>
<td>20.0</td>
<td>0.1</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>12</td>
<td>20.0</td>
<td>0.1</td>
<td>2</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>13</td>
<td>20.1</td>
<td>0.0</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>14</td>
<td>20.1</td>
<td>0.0</td>
<td>2</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>15</td>
<td>20.1</td>
<td>0.1</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>16</td>
<td>20.1</td>
<td>0.1</td>
<td>2</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>17</td>
<td>20.0</td>
<td>49.9</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>18</td>
<td>20.0</td>
<td>0.0</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>19</td>
<td>20.0</td>
<td>-0.1</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>20</td>
<td>20.0</td>
<td>0.0</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>21</td>
<td>20.0</td>
<td>0.0</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>22</td>
<td>20.0</td>
<td>50.0</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>23</td>
<td>20.0</td>
<td>-0.1</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>24</td>
<td>20.0</td>
<td>0.0</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>25</td>
<td>20.0</td>
<td>0.0</td>
<td>0</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>26</td>
<td>19.9</td>
<td>49.9</td>
<td>2</td>
<td>no</td>
<td>&lt;</td>
</tr>
<tr>
<td>27</td>
<td>25.5</td>
<td>25.5</td>
<td>1</td>
<td>yes</td>
<td>==</td>
</tr>
<tr>
<td>28</td>
<td>20.0</td>
<td>0.0</td>
<td>1</td>
<td>no</td>
<td>&gt;</td>
</tr>
<tr>
<td>29</td>
<td>20.1</td>
<td>20.1</td>
<td>0</td>
<td>no</td>
<td>==</td>
</tr>
<tr>
<td>30</td>
<td>19.9</td>
<td>19.9</td>
<td>1</td>
<td>no</td>
<td>==</td>
</tr>
</tbody>
</table>

Table 5.5: Value Combinations Considering Comparisons

- **Step 6:** Finally all invalid value combinations which contain invalid values of one or both of `maxPrice` and `minPrice` are analysed (Item 23 to Item 35 in Table 5.5), together with all comparative relationships. Also `minPrice == maxPrice` is not covered, so one new invalid combinations is generated and added as the last item in Table 5.5. In the new added combination the values of `maxPrice` and `minPrice` are not both valid. The final set of value combinations has 39 items.

The current testing tool supports the user to specify comparative statements in the format of plain text, so the user can only specify an expression with basic math operators (i.e. +, -, *, and /). However, the editor of comparative statements can be improved in the future so that more complicated math expressions can be specified and comparisons between them can be taken into account in test case generation.
CHAPTER 5. TEST INPUT GENERATION

5.4 Variable Assignment

To execute test cases, their input values must be assigned to the implementation of associated variables at runtime. The technique for assigning a value to a variable may vary depending on how the variable is coded in the implementation of the system. For example, a variable that is private to an object \(^{12}\) needs to be set via the object’s mutator functions. Because the testing process is fully automated, it is necessary that there be some specification of a matching relation in the design documentation to allow appropriate assignment of variable values to the implementation of the variable in the code.

This matching relation is specified in the unit test descriptor of the unit and takes the form \(<\text{variable-name}, \text{type}, \text{assignment}>\) for each variable, where the type is classified as simple, complex, belief or function based on how the variable is implemented in the source code of the system. The process of value assignment depends on these types as described below.

**Simple variables**

A simple variable is implemented as a public variable or a private variable that is set via a public mutator function. For example, in Figure 5.8 the variable \(\text{BookID}\) is implemented as the public variable \(\text{bookID}\) in the event class. Hence the assignment is a direct value assignment.

The variable \(\text{BookName}\) is implemented as a private variable \(\text{bookName}\) in the same event class with a public mutator function \(\text{setBookName(String)}\). The value assignment is therefore via this function, which is specified in the unit test descriptor (see the system design document in Figure 5.8).

In general, if no assignment relation is specified for a variable, the assumed default is a simple variable that is publicly accessible.

**Complex variables**

A complex variable is one that is implemented as part of a nested structure, such as an attribute of an object, which may in turn be part of another object and so on. For example, in Figure 5.9, the variables \(\text{Email}\) and \(\text{Name}\) are attributes of the \(\text{attd}\) object of type

\(^{12}\)In terms of Object Oriented Programming.
Attendee in the triggering event of the plan. The assignment relation for the Email variable is `att.email` as it is a public attribute of that object and can be set directly. The Name however, has to be set via the `att.setName(String)` method.

Belief variables

These are variables that are fields of a particular belief-set. For example:

```
[belief-var, {StockDB, stock.dat, bookID}, int, {==1}]
[belief-var, {StockDB, stock.dat inStock}, int, {>0 , <5}]
[belief-var, {StockOrders, order.dat, bookID}, int, {==1}]
[belief-var, {StockOrders, order.dat, required}, int, {>0, <=30}]
```

If the variable is a belief variable then no assignment function is required in the design as the technique for assigning variables is the same for any field of the belief. That is, create and insert a record with the values generated for the belief variables of concern and random valid values for the rest of the fields for that record. For example, in Figure 5.10 the first test case
"SD" and "SO" are the abbreviated names of “StockDB” and “StockOrders” respectively and “random” indicates that the value is generated randomly but should be valid.

Figure 5.9: Illustration of Complex Variables

Figure 5.10: Value Assignment of Belief Variables
CHAPTER 5. TEST INPUT GENERATION

is set up by creating and inserting a record for the *StockDB* belief with fields *bookID* and *inStock* set to 1 and 5 respectively (the boundary values) and creating and inserting a record for the *StockOrders* belief with fields *bookID* and *required* set to 1 and 30 respectively.

Although the automated test cases contain randomly generated values for the belief fields that are not specified as unit test variables, as with all the test units the user may specify manual test cases (value combinations) prior to executing the tests.

A special case of belief variables is when historical information is required. For example, the context condition of the plan may contain the following:

\[
\text{StockDB.getStockAt}(t_1) > \text{StockDB.getStockAt}(t_2)
\]

which is relevant to two historical values of the same variable. The design document will specify the relevant variable as:

\[
[\text{belief-var}, \{\text{StockDB, stockdat, numberInStock}\}, \text{int}, \{>0\}]
\]

Following the approach above, value combinations will be generated for *numberInStock* but only one record will be inserted per test case, which would be insufficient to evaluate that particular context. In general, this situation is where the belief is to be populated with multiple rows. There are two alternatives for testing such situations in the testing framework: the first is for the user to specify a test case manually for this belief. When specifying a test case manually for a belief, the user is given the option to add as many rows as desired. This approach however could be tedious if many rows are to be inserted. The second approach is to provide a method for populating the database as an initialization procedure of the unit under test as discussed in section 4.3.3.

**Function variables**

There may be instances where a variable in the design is realised by a function in the implementation. Such a variable is called a *function* variable. For example, a variable *total order cost* that indicates the total cost of an order may require some calculation. It is not possible to set the value of this variable as it depends on the value returned by the function. This value may depend on a number of other variables, some local to the function and others external. The current testing framework ignores such variables when generating test cases.
One way in which function variables could be tested is for the user to specify all non local variables within the function that determine the return value in the design document, then value combinations are generated for these variables. Another approach would be to augment the system source code to replace the call to the function by another variable whose values can be set for testing purposes.

5.5 Specification of False Positives

The testing tool allows the user to specify that a fault detected is a false positive. With the specification, the fault will no longer be reported on the subsequent testing of the same system. The specification for a fault as a false positive consists of three parts:

1. **Index of the fault type**, e.g. FT\_COMPLETION, FT\_CC\_VALID

2. **List of the associated units** (optional)
   These are the units that are associated with the false positive. For example, if a fault of FT\_OUTEV\_NEVER (some specified outgoing events are never posted) detected is associated with multiple outgoing events, the user can specify that the fault is a false positive for some of these events.

   This part is ignored if the fault type is not associated with any other units (e.g. FT\_COMPLETION, FT\_CC\_VALID), or if the fault is a false positive to all associated units (e.g. non-posting of all events is a false positive in the above example).

3. **List of the associated test cases** (optional)
   These are the test cases in which the fault is a false positive. For example, if a fault of FT\_COMPLETION (the plan failing to complete) is detected in case.1, case.2 and case.3, the user can specify that in case.1 and case.2 the fault is a false positive.

   This part is not needed in two situations: the fault is detected across all test cases (e.g. FT\_CC\_VALID: the context condition of the plan under test is always evaluated as true) and the fault is a false positive in all test cases.

Figure 5.11 shows an example of false positive specification for a plan.\(^{13}\) The example shows three false positives: the plan failing to complete in case.1 (FT\_COMPLETION), the “BookQuery” event not posted (FT\_OUTEV\_NEVER) and the context condition value

\(^{13}\)Currently the user must manually specify this as plain text, but it is straightforward to provide the interactive user interface for such specification, as the testing tool is being improved.
always being true (FT_CC_VALID). A test case is specified in form of a combination of input variable values.

**Summary**

This chapter discussed the approach of generating test case inputs in the testing framework, based on the information of input variables declared by the developer in the design descriptor of the unit under test. The comparisons between variable values are also taken into consideration. The values of input variables are automatically assigned to the implementations of variables at runtime, as part of the automated testing process. We have now introduced and discussed all the aspects of the automated testing framework. The next chapter presents the evaluation of the testing framework.
Chapter 6

Evaluation

In order to evaluate the effectiveness of the testing framework, experiments were carried out to ascertain the ability of the testing framework to discover problems in a system under test. In the experiments, a series of sample systems developed by postgraduate students were tested and test results were analysed in order to evaluate the testing framework.

In this chapter, the issues to be addressed during the evaluation are discussed in section 6.1, followed by the experimental process (section 6.2) and the application for evaluation (section 6.3). In the experiments, the faults detected are investigated to evaluate the effectiveness of the testing framework. The investigation and analysed results are discussed in section 6.4. Section 6.5 presents examples of some faults exploring the kinds of problems that were detected. Finally, some other aspects of the evaluation are discussed in section 6.7, such as how annotation details affect the testing results of a system.

6.1 Evaluation Objective

The evaluation objective is to check if the faults detected effectively reveal problems that exist in the design and the implementation of a system. Two issues are raised relevant to the effectiveness of the testing framework as follows:

- Is each fault detected sensible?
  Ideally each fault detected should indicate a problem that exists in the system. To verify this, each fault detected in the experiment has been investigated to discover the cause that leads to the fault. The investigation reveals different kinds of problems indicated by the faults detected. For example, some faults indicate problems in the
implemented code of the system, while some other faults indicate something missed in the design.

Although the fault types defined in the fault model (refer to section 3.2) describe situations in which problems may exist in an agent system, it is still possible that a fault detected is due to some implementation specific situations that are not actually problems. For example, a plan fails to complete due to time-out when it is tested, because the plan terminates the execution of the system program\(^1\). This is not a problem because the plan is developed for terminating the program in some particular situations. Such situations are regarded as false positives.

- Are there some fault types that we have defined that never actually occur?
  
  We also investigate if there are some fault types that have been defined in the fault model that actually never occur, in order to evaluate if all the fault-types we have defined are sensible.

In order to addresses these two issues, a series of agent systems have been tested and the faults that were detected have been analysed. The following section introduces the experimental process in which these systems were tested.

### 6.2 Experimental Process

Before the evaluation is presented, we have done preliminary evaluation of the testing framework in order to reveal and fix errors in the testing tool itself and make some adjustments. We used an agent system that had been developed by a research assistant as an exemplar for use in tutorials and workshops, to illustrate agent oriented design using Prometheus. This system was a simplified version of a weather alert agent system developed in collaboration with the Bureau of Meteorology Australia. The preliminary evaluation was conducted following the same experimental process as for the primary evaluation presented below. The final testing results of this system are shown in Appendix D and part of the test report is shown in Appendix E.

In the experiments for the primary evaluation thirteen agent systems were tested. Each system implements a client application that communicates with a server. These systems were developed by different postgraduate students. Each system was developed with a design specification generated using PDT [Padgham et al., 2008a] and an implementation developed

\(^1\)In terms of Java programming, the plan invokes “System.exit(0)” to terminate the system program.
using the JACK agent programming language [Winikoff, 2005]. The application implemented by the systems is introduced in section 6.3.

It is difficult to collect commercial applications for evaluation work as companies are usually not willing to release their source code. Also no other JACK applications were available from outside our research group.

The experimental process for a system consisted of three steps as follows:

1. Setup
   The system was annotated with testing specific information in the design descriptor (e.g. declaration of input variables (see chapter 5)) and any code needed for initialisation procedures (see section 4.3.3) was implemented. Some systems have been annotated by the developers themselves.

2. Execution
   Each system was automatically tested by the testing tool, following the global testing process presented in section 4.1. The test harness for each unit was automatically generated and test cases were automatically generated and executed based on the declaration of input variables. Appendix C shows screenshots that represent how a system is tested by the testing tool that has been integrated into PDT. All the faults detected are described in the test report of the system.

3. Fault Investigation
   After a system was tested, each fault detected was investigated to discover the problem that leads to the fault, by reviewing the design descriptor and the implemented code of the associated unit, and if necessary the rest of the system. The fault may be caused by a problem in the design or in the implementation, or may be due to a particular situation that is a false positive. The cause of each fault was logged for further analysis and evaluation.

   Some faults detected were due to incompletely or incorrectly annotated information. For example, an event variable accessed by a plan was not specified as an input variable, leading to an exception when the plan was tested, as the variable was not assigned values. In such a case, the annotated information was improved and the associated unit was tested again until there is no fault due to incomplete or incorrect annotation.

   Each system required between about 2 to 4 rounds of annotation to eliminate these problems. More effort was required in the earlier systems and this decreased as expe-
rience in annotation was gained. In four systems students did volunteer to annotate their own code. However, as they were inexperienced in doing annotation, such as what variables should specified as input variables in test descriptors, these systems also required some updating before successful testing could occur. In summary, quality of annotation depends on both understanding of our testing framework (at least partially, e.g. test descriptors) and understanding of the system.

During the investigation of some faults and the problems revealed by the faults, we have also found out additional situations in which similar problems may exist but were not detected by the test cases generated. Based on these situations, we analyse limitations of test input generation and consider possible improvements as discussed in section 6.5.

All the investigation results were collected together and analysed to investigate the underlying cause of each fault detected. The investigation and analysis details are discussed in section 6.4.

6.3 Application Systems for Evaluation

The systems used for the evaluation were developed by postgraduate students in a course on agent programming. These systems implement the same application, but the details of the design and the implementation vary as they were developed by different students. Each system implements a client-server based application that simulates a scenario of gold mining, which is the scenario of the Agent Contest 2007 \(^2\). Each system is a client program that receives data from the server, simulating a process in which gold is looked for and picked up in a given area.

Figure 6.1 shows the structure of the application. The client program has a communication component that communicates with the server. The communication component, after receiving information from the server, creates an event that carries the information and sends out the event to trigger the associated plans in a Player agent to perform the scenario. The communication component also receives events sent by plans in a Player agent when the agent needs to send an action request to the server.

Each system is developed based on the skeleton code of the client application that consists of the communication component and the code of other basic classes. The agents and the units within these agents were developed by students according to an instruction document.

\(^2\)http://www.multiagentcontest.org/2007

143 (October 14, 2011)
Therefore these systems have different units (plans/belief/events) although some of them may perform similar functionalities. Each system has between 1 - 4 agents types and between 30 - 80 units. Therefore there is reasonable variety on these systems.

Before each system is tested, the server program is started up to ensure that the plans that require communication with the server work properly.

6.4 Fault Analysis and Evaluation

All the faults detected in the experiments and their causes have been analysed to determine whether the faults detected indicate actual problems and whether some fault types defined were never observed. All faults detected are categorised according to different kinds of problems indicated by the faults (e.g. problems in the design, or problems in the implementation). Then the distribution of different categories of faults is calculated and analysed. We also analyse how different fault types contribute to faults of different categories.

Fault Categorisation

We refer to the work of Boehm [2005, page 307] and Ramberger et al. [2004] for fault categorisation in the scope of unit testing. The former categorises faults as commission faults that indicate something incorrectly developed, and omission faults that indicate something missed in the design or in the implementation. The latter defines four fault categories as follows:

- coding faults: the mistakes made in the implementation
CHAPTER 6. EVALUATION

- documentation faults: the mistakes in the design document, e.g. the parameters of a method are incorrectly specified.

- Incomplete coverage: all faults caused by the fact that certain code lines could not be executed.

- Other faults

In our approach of fault categorisation, faults are categorised based on: (I) if they are caused by actual problems or not, and (II) if they are caused by problems in the design or in the implementation. Taking into account the categorisation of commission faults and omission faults. The categories are defined as follows:

1. **incomplete implementation** is a coding fault of omission:
   A fault of this category is detected because a feature that has been specified in the design is not implemented.

2. **incorrect implementation** is a coding fault of commission:
   The internal logic of the associated unit is not correctly implemented.

3. **incomplete design** is a documentation fault of omission:
   A feature that has been implemented is not specified in the design.

4. **unclassified mismatch**
   A feature specified in the design is not consistent with the implemented code. We cannot confirm if such a mismatch is due to a design problem or an implementation problem, so it has been categorised separately.

5. **false positives**
   Some faults detected are not due to actual problems when investigated.

6. **redundant faults**
   We categorise a fault as redundant if it was caused by problems that had earlier been identified by another fault.

It is difficult to evaluate our testing framework by experimental comparison of our approach with other approaches for agent testing, because that requires testing of the same systems using some other approach and comparisons of testing results. There may be two possible strategies but both of them are hard to realise. One is to re-test those evaluation
systems using some other testing tool (e.g. eCAT [Nguyen et al., 2010]), which however is usually based on a different AOSE methodology and development platform (e.g. eCAT based on the Tropos methodology and the JADE platform). In the case of eCAT, those evaluation systems need to be re-implemented with Tropos and JADE so that eCAT can test them. Such re-implementation may introduce new faults that can obviously affect testing results. The other strategy is to implement the techniques of some other approach with Prometheus and JACK then those evaluation systems can be directly tested. That re-implementation however is not straight forward so is difficult to perform. Consequently we instead analyse the results obtained with reference to frameworks developed within conventional software testing (i.e. the work of Boehm [2005, page 307] and Ramberger et al. [2004]) to evaluate our approach.

6.4.1 Fault Distribution

![Figure 6.2: Distribution of Faults](image)

Figure 6.2 shows the number of faults in each categories. False positives have been split into those generating warnings and those generating errors. As can be observed:

- Most of the faults detected (over 96%) can be clearly categorised with exact reasons, with less than 4% being unclassified mismatches.

- Around 62% of the faults detected reflected underlying problems of some category,
while over 38% were false positives.

- Incomplete design contributes around 1/3 of all detected faults, reflecting the issue that developers frequently implement more than what has been specified and forget to update the design.

### 6.4.2 Fault Categories and Fault Types

For each of the categories, we have examined which fault types from the fault model are responsible for generating these faults (see Appendix F, Table F.1). Based on the results, faults of each category are analysed in detail for summarising problems of each category. This section discusses the categories of actual problems (excluding false positives) and their relevant fault types. False positives are discussed in the next section. Although all false positives can be removed in subsequent testing of systems after they are manually identified and noted as not being actual problems, we have still investigated these false positives to analyse if possible improvements can be applied to automatically avoid them.

**Incomplete Implementation**

Plans seem to be more problem-prone than events and beliefs in implementations. This can be supported by the fact that most faults of incomplete implementation (68/84, refer to Table F.1) and incorrect implementation (12/18) are related to plans.

The most common faults of incomplete implementation are specified outgoing events not posted, either from plans or from beliefs((55+16)/84), due to the reason that the code for event posting is not implemented. In the remaining 13 faults, 6 faults of the context condition (CC) value always being false (FT\_CC\_INVALID) are associated with plans that were not completely implemented yet, so their context conditions were temporarily implemented as “false” \(^3\) to make these plans not executable. 6 faults of the CC value always being true (FT\_CC\_VALID) are due to context conditions specified not being implemented. This would seem to be simply that the developers omit to implement something that has been specified in the design. The other fault is associated with a belief that posts an event that has no handling plan implemented.

\(\text{\textsuperscript{3}}\)The context condition method is implemented as “{false;}” in JACK.
CHAPTER 6. EVALUATION

Incorrect Implementation

18 faults of incorrect implementation are due to problems in context conditions and bodies of plans. The former has 6 faults, while the latter contributes 12 faults of plans that fail to complete.

Incomplete Design

In 144 faults of incomplete design, all but 4 are due to two reasons: absence of context conditions of plans in the design (107) and overlap or incomplete coverage not specified occurring for an event (11+22). This may reflect the fact that the developers easily omit the specification of context conditions and coverage/overlap.

The other 4 faults are associated with specified events without either handling plans or plans that post them in the design. None of these 4 events is a percept 4 or an event from an external agent, so each of them should at least have a plan that handles it and a plan that posts it.

Unclassified Mismatches

There are a few faults of unclassified mismatches (18). They are all associated with plans having different triggering events or sub-plans in the design and in the implementation. This may reveal a problem that the design and the implementation were not updated synchronously during the development.

Redundant Faults

For the redundant faults detected, 11 of them are actual problems. The other 13 are false positives and will be discussed later in section 6.4.3. In the 11 actual problems, 7 of them are exceptions thrown by plans. These exceptions are caused for the same reason as 7 faults of plans failing to complete in the category of incorrect implementation. When an exception is thrown by a plan’s body, that plan always fails to complete. An exception reported usually describes the associated problem more precisely than a fault of plan failing to complete. Hence an improvement can be applied to fault identification that if an exception is thrown by the body of the plan under test, the failure of plan completion will not be reported.

4an external input to the system in terms of Prometheus
CHAPTER 6. EVALUATION

The other 4 redundant faults are absence of cyclic executions (FT_CY_EXISTENCE). One of them is because the associated cyclic plan is never executed due to the reason that its context condition is implemented as “false”. This has also caused a fault of the CC value always being false (FT_CC_INVALID) categorised as an incorrect implementation. The other fault is because the associate cyclic plan never posted a specified event that is a part of the specified cyclic structure. The same problem has been revealed as a fault of a specified outgoing event not posted (FT_OUTEV_NEVER) categorised as incomplete implementation.

6.4.3 False Positives

179 faults detected are false positives (60 errors and 119 warnings). Although they will not be reported as faults on subsequent testing after being identified and noted, we would of course prefer that false positives are not reported as faults in the first place. Therefore these false positives have been investigated to analyse if they can be automatically avoided to improve the effectiveness of the testing framework. There are three different situations. For some of them, the associated fault types should perhaps be excluded from the fault model as discussed below, because faults of these types are only amongst false positives in our evaluation. For some others possible improvements of the testing framework have been considered to avoid these false positives. These improvements are discussed in this section and may be implemented as future work. The rest of the faults are difficult to automatically avoid and only manual examination can identify them as false positives. However, for some of them, if the design model of a system can provide extra information, they may be automatically avoided. These three situations are discussed in detail in the following:

Fault Types That Could Be Excluded

21 false positives (warnings) are identified as overlap (FT_EV_OVERLAP_W, # 2) or incomplete coverage (FT_EV_INCOMP_W, # 19) that have been specified for associated events, and also occur in some test cases. These two fault types could be excluded. However, investigations of more agent systems are necessary to determine if this is warranted, given that unintended overlap or incomplete coverage is a common cause of problems [Padgham et al., 2005, page 22].

5Further evaluation should however be carried out before such a decision is made.
CHAPTER 6. EVALUATION

**Improvements to Avoid False Positives**

22 false positives could be removed by improvements in code augmentation or test input generation. 5 of them reflect two limitations of test input generation. The first one is the omission of the variables used in the associated meta-reasoning plan when an event is tested. This limitation is revealed by 1 warning of non-execution of a plan that handles the event under test (FT\_EV\_NOT\_EXEC). There is a meta-reasoning plan implemented for the event under test. The meta-reasoning plan does select the plan associated with the warning for execution when a particular condition is satisfied. However, variables used in the meta-reasoning plan are not included as input variables of the event under test. Hence in all event test cases the condition for executing the associated plan was never satisfied. This warning can be removed if input variables for testing an event also include the variables used in the associated meta-reasoning plan.

The second limitation is that combinations of comparison relationships between input variables are not considered in test input generation, such as “x==dx & & y<dy” in the context condition of a plan. Test input generation has considered the coverage of each single comparison relationship such as “x==dx” or “y<dy” (refer to section 5.3), but the coverage of their combinations are not taken into account hence cannot be guaranteed in the test inputs generated. Consequently the context condition value of a plan, if a combination of comparison relationships is involved, may never be true in any generated test case. This limitation contributes 1 false positive of the CC value always being false (FT\_CC\_INVALID), 1 false positive of a specified outgoing event not posted (FT\_OUTEV\_NEVER) and 2 false positives of non-execution of plans handling the event under test (FT\_EV\_NOT\_EXEC). An improvement of test input generation to address this limitation has been considered and will be discussed later in section 6.6, when we discuss some examples of the faults detected. The false positives associated with this limitation may be removed with the improvement, but this can only be ascertained after the improvement is actually applied in the future.

The other 17 false positives can be removed by improvements in the code checking or code augmentation process of each associated unit. They are:

- 1 warning is a specified outgoing event not posted (FT\_OUTEV\_NEVER). The event is posted in the fail method that in JACK is invoked only when the plan fails. It would be possible to check for this situation by examining the plan’s code and not reporting a fault if the event not posted is generated only from the fail method.
• 16 warnings are implementations of attempt-triggered callback methods (FT\_BE\_INAPPRO\_CB). This fault type has been defined to report the risk that an attempt-triggered may post the event of the action rule to be verified when the operation of the action rule fails. However, all attempt-triggered methods associated with these warnings are empty without any logic implemented, so there is no risk that events of action rules will be posted. The code augmentation for a belief can be improved to check if each attempt-triggered method is empty associated with the action rule to be verified. If all associated attempt-triggered methods are empty, a fault of FT\_BE\_INAPPRO\_CB need not be reported.

Although all faults of this type detected in our evaluation are only amongst false positives, this fault type should still remain. This is because a developer may implement event posting in an attempt-triggered method, leading to a problem that an event may be posted when the operation of the associated action rule fails.

**False Positives Not Avoidable**

136 false positives that cannot be automatically avoided are:

• 2 warnings are outgoing events not posted (FT\_OUTEV\_NEVER). These events are implemented as a more generic event with the specialisation indicated as a parameter. Consequently they cannot be automatically recognised. For example a drop event is implemented as an executeclima action with a parameter “drop”.

• 2 warnings are absence of cyclic executions (FT\_CY\_EXISTENCE). Manual examination is necessary for checking if the associated plan’s logic actually implements cyclic executions or not.

• 62 warnings are associated with components that have been implemented but are not specified in the design for particular reasons. 36 of them are non-specified events being posted at runtime (FT\_OUTEV\_NOT). These events are implemented for debugging purposes, so it is not necessary to specify them in the design. If the design documentation allows tagging of the components that are developed for purposes of debugging, these results could be avoided.

The other 26 are specified outgoing events not posted (FT\_OUTEV\_NEVER), due to the reason that the handling of these events was not automatically observed. The plans

151 (October 14, 2011)
that handle these events in the implementation are not specified in the design, because these plans are imported from external libraries in the implementation. However, if the design model provides specification of external components used by the system, such as structures of external libraries imported, these false positives may be avoided. In addition, the problem of external libraries also lead to 13 redundant faults when the associated events were tested. These faults are events without handling plans in designs, as detected by code checking.

- 10 warnings are context condition values always being true in all test cases (FT_CC_VALID), because specified context conditions are not implemented. These context conditions specified seem to be relevance conditions of associated plans. They are false positives because PDT does not provide an entity for the specification of a relevance condition, so students specified the relevance condition of a plan into the context condition entity of the plan’s design descriptor.

- 60 errors are plans failing to complete (FT_COMPLETION). Some of these plans return false in particular situations and the other plans terminate the execution of the owner agent or the system in particular situations. These false positives cannot be avoided as manual examination is necessary to identify if these situations are actual problems or not. However, the code augmentation for a plan can be improved to provide more precise notification if such a fault is detected, to facilitate fault investigation. Particular test code can be inserted that observes the execution of the termination code or the returning code in the plan’s body, using the same mechanism as that for observing the posting of an outgoing event (refer to section 4.4.3).

After investigation of all false positives, over one third of them (69) may be automatically avoided with exclusion of unnecessary fault types (21), possible improvement in the testing framework (22) and extra information provided by the design model (26). The other cases (110) cannot automatically avoided and manual examination is necessary. However, 60 of

6If a plan test harness cannot extract from the design the sub-plans that handle a particular outgoing event, the harness cannot observe the handling of the event at runtime after the event is posted and will identify that the event is not successfully posted, refer to section 4.4.1 for details.

7There are 13 associated events of the 26 FT_OUTEV_NEVER faults, as some faults are associated with the same event.

8A relevance condition of a plan is a filter of the type of the event handled by the plan via “getAgent().finish();” in terms of JACK

9via “getAgent().finish();” in terms of JACK

10via “System.exit();” in terms of Java

152 (October 14, 2011)
them can come with more detailed fault descriptions if an improvement in code augmentation is applied.

### 6.4.4 Testing Process Evaluation

In the testing process discussed in section 4.1 all the units within each agent are tested in an appropriate order, which is determined according to dependencies between units. The testing process is *interrupted* when an error or exception is detected in a unit so that all the units (*dependers*) that depend on this unit (*a dependee*) will not be tested until the error/exception is addressed. Hence there will often be multiple testing cycles for a system until all errors/exceptions are addressed. In our experiments each evaluation system was tested for 3 to 7 cycles.

An alternative approach is a *non-interrupted* testing process in which a system is tested without pausing to fix any error or exception. In this case all the units of a system will be tested in one cycle, but redundant faults caused by problems of dependee units will be reported on and must be sorted out during analysis. A non-interrupted testing process for each evaluation system usually has 2 to 3 testing cycles. All exceptions/errors were detected in the first cycle, then they were addressed and the system is tested again to confirm all exceptions/errors have been removed. Sometimes an extra cycle was needed because some errors/exceptions may be omitted.

This non-interrupted testing process has also been carried out for each system used for evaluation, in order to assess the necessity of not testing dependee units until problems in their dependee units have been fixed, and to evaluate the relative efficiency of these two approaches.

**Necessity of Fixing Dependee Problems**

The faults detected in the non-interrupted process of each system have been compared with the faults obtained in multiple test cycles of the interrupted process of the same system. The comparison results show that faults of some error types and some warning types caused faults in dependee units, while faults of some other error types did not. All the exceptions detected are redundant faults of plans failing to complete (*FT_COMPLETION*), so the exceptions are discussed together with these errors. All the faults that lead to dependee faults are of two error types and one warning type:

- The most common faults leading to dependee faults are plans failing to complete
(FT\_COMPLETION), which caused faults of the same type in all depender units.

- The incomplete coverage of a subtask event led to the completion failure of a plan that posted the event, as there was no sub-plan applicable for the posted subtask event.

- A warning of the CC value always being false (FT\_CC\_INVALID) is because the plan’s context condition was implemented as “false”. This problem also caused 2 extra faults when the event handled by this plan was tested. The first fault is the non execution of that plan as its context condition is always false. The second fault is incomplete coverage of the event under test, as that plan is the only one that handles the event and the plan is never applicable due to its context condition value.

In order to avoid a substantial number of redundant fault notifications, it is necessary that depender units are not tested until the faults of the above types detected in dependee units have been fixed. On the other hand, the errors of FT\_CC\_ABSENCE (absence of CC in the design) and FT\_EV\_OVERLAP\_E (overlap does exist - does not matches design) never caused extra faults in depender units when detected. Hence an interrupted testing process could be improved to continue when errors of these two types are detected, instead of stopping the testing of depender units. Actually over half of the errors detected in the systems tested for evaluation are of these two types (129/233). If no depender units are skipped for testing when these errors are detected, the number of testing cycles for a system could be reduced. The evaluation systems were tested for 3 to 7 cycles until all units had been tested. If errors of these two above types are ignored, there will be 2 to 5 cycles for these systems. Table 6.1 shows the original number of test cycles for each system and the number after these two error types are ignored.

We could also provide a more flexible functionality so that the user can specify particular fault types which if such faults are detected in a dependee unit, dependers will not be tested. By doing this, the user can determine what fault types may affect dependers according to particular applications.

In addition, the testing process could also be improved in relation to the occurrences of exceptions. Because we are doing unit testing there are no exceptions relevant to interactions between agents. Hence when a termination with exception is encountered, all units in the other agents can still be tested. This improvement could also reduce the number of testing cycles for a system.
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>system index</th>
<th>#. test cycles</th>
<th>#. test cycles (ignoring two error types)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 6.1: Numbers of Test Cycles

Time Efficiency

The time it takes for testing a system depends on how many testing cycles occur for the system and how long the system is tested in each cycle. The number of testing cycles of the interrupted testing process for a system depends on the number of errors/exceptions detected and the complexity of the hierarchical structure of the agent under test. A non-interrupted testing process may have less testing cycles than an interrupted testing process for the same system. However, much time has to be spent for manually distinguishing those faults that are caused by problems in dependee units. It is time consuming especially when faults caused by dependees are detected in many test cases. In our experiments the non-interrupted testing process for an evaluation system usually spends more time than the interrupted testing process for the same system. Therefore it is still recommended to test a system using an interrupted process as discussed in chapter 4.

The overall time for testing a system in one cycle depends on both the hardware environment and the number of test cases. Table 6.2 shows some examples of time cost for systems with different numbers of units and test cases. Usually the time cost for a system increases linearly with the increase in the number of test cases, which can be controlled with the setting of the number of input variables and the value ranges specified, options of variable values (minimal, normal and comprehensive levels), options of value combinations (basic and extended levels), and the maximum limit of the number of test cases (refer to section 5.2).

---

These examples were run on a dedicated PC with an Intel Core2 Duo E8500 3.16 GHz processor, 4 GBytes of RAM.
One approach that could be used to reduce the time cost for testing a system is the specification of particular units that do not need to be retested. When an error/exception is fixed and the system is tested again, those units that have been tested before are always tested again the following time. The user could specify such units as not needing to be tested to reduce the time cost. However, such specification should be done very carefully as fixing of errors/exceptions may also affect the units that have already been tested.

As a second possible time saver, we could adjust the time out mechanism used to control the execution of a test case. The test agent will terminate the current test case if it does not receive any response from the system after a period of a specified time-out value is elapsed. It could take a long time to test a system if time-out frequently occurs in test cases. Shortening of the time out value used could substantially reduce testing time.

6.4.5 Coverage of Fault Types

Of all 21 fault types defined in the fault model (refer to section 3.2), 12 of them are observed as actual problems, 3 of them are observed only amongst false positives and the remaining 6 are not observed. The 9 (6+3) fault types that are never observed as actual problems are summarised in Table 6.3. In these fault types, 6 of them are related to beliefs, 2 are related to events and the other 1 is related to cyclic plans. This distribution may indicate that beliefs are less problem-prone than events and plans.

2 of these 9 fault types could perhaps be removed from the fault model. They are FT_EV_OVERLAP_W (overlap does exist - matches design) and FT_EV_INCOMP_W (coverage does not exist - matches design) that have been discussed above in the analysis of false positives.

We believe that the other 7 fault types should still remain as they do indicate possible problems in agent systems.
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>Observation</th>
<th>Fault Type</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>not observed</td>
<td>FT_CY_NO_STOP</td>
<td>exceeding of cyclic execution iterations</td>
</tr>
<tr>
<td></td>
<td>FT_BE_NO_FIELD</td>
<td>a specified data field not implemented</td>
</tr>
<tr>
<td></td>
<td>FT_BE_NOSPEC_FIELD</td>
<td>an implemented data field not specified</td>
</tr>
<tr>
<td></td>
<td>FT_BE_WRONG_FIELD</td>
<td>inconsistency of a data field in the design and in the implementation</td>
</tr>
<tr>
<td></td>
<td>FT_BE_NO_CB</td>
<td>absence of callback methods implemented</td>
</tr>
<tr>
<td></td>
<td>FT_BE_WRONG_POST</td>
<td>posting of an event when a belief operation fails</td>
</tr>
<tr>
<td>only amongst false positives</td>
<td>FT_EV_OVERLAP_W</td>
<td>coverage does not exist - matches design</td>
</tr>
<tr>
<td></td>
<td>FT_EV_INCOMP_W</td>
<td>overlap does exist - matches design</td>
</tr>
<tr>
<td></td>
<td>FT_BE_INAPPRO_CB</td>
<td>implementation of attempt-triggered callbacks</td>
</tr>
</tbody>
</table>

*refer to section 3.2 for full description

Table 6.3: Fault Types Not Observed as Actual Problems

6.5 Further Analysis of Selected Faults

This section discusses some example faults and the problems revealed by them. For each fault, we will introduce the functionality of the associated unit under test, the problems that exist in the unit (in the design or in the implementation) and the faults detected by the test cases due to unit problems. Particularly, for some faults that are categorised as incorrect implementation, we have investigated them in detail to check whether the implementation problems revealed do cause system errors when the associated systems are executed in certain situations and what system errors they are. Based on the investigation results, we discuss how the faults detected help to reveal problems in a system and to improve the robustness of a unit under test.

Six examples in different evaluation systems are discussed in this section. The first example presents a mismatch between the design and the implementation. The next four examples are completion failure of a plan. Three of them were caused by underlying implementation problems that also lead to system errors. Although the fifth one does not cause system errors, the fault detected still indicates a potential problem of the associated unit under test and addressing this can help to improve the robustness of the associated unit. The last example presents a redundant plan that is never executed.

In the investigation we have also found additional situations that had not been discovered by the test cases generated, but in which problems may also occur. When these additional situations were analysed we have revealed a limitation of the algorithm of test input generation and an associated improvement is discussed in section 6.6.
CHAPTER 6. EVALUATION

Fault 1: Incomplete coverage of the event under test.

A fault of incomplete coverage was detected for an event due to the mismatch of the plans that handle the event in the design and in the implementation. The event $EAct$ is posted to carry out an action of the agent. The event is handled by 4 plans in the implementation: $MoveToDepot$, $FindGold$, $PickupGold$ and $MoveRandomly$, one of which will be activated for execution. However, in the design only two of them are specified as handling the event ($MoveToDepot$ and $MoveRandomly$) and the other two are specified as handling other events.

In some test cases of the event, the two plans specified in the design were not applicable due to their context condition values. In these cases one of the plans not specified in the design would have been applicable (e.g. $FindGold$), but was not executed because it was not included in the subsystem under test\footnote{Only the units that belong to the subsystem under test are allowed for execution, refer to “Subsystem under Test” in section 4.3.1 for details.}, due to the reason that the plan was not specified in the design as a plan that handles the event. Consequently there was no plan executed and a fault of incomplete coverage was reported on with the underlying problem being incomplete design.

Fault 2: The plan under test fails to complete.

The plan $GoToGold$ looks for a path from the current position of the agent to each gold in a search grid, then determines the direction of the next step of the agent. The plan will throw an exception if it cannot find a path to some gold in the search grid.

Figure 6.3 shows an example of a failure situation. The plan fails to find a path to a gold that is surrounded by obstacles (in the bottom) and throws an exception.

This problem can cause an error that the agent stops moving when the associated system is executed. The task of an agent in the system is to look for and pick up gold in the search grid, then drop off gold to a depot. The agent executes multiple iterations to collect gold in the search grid. In each iteration, the agent posts an event $EAct$ to activate a plan\footnote{The event is handled by multiple plans.}, which decides a moving step of the agent in the search grid (e.g. towards a gold) and then makes the agent move to the new position. $GoToGold$ is one of the plans handling the event and is activated when there is known gold in the search grid. When the plan fails due to the exception, no action is scheduled for the agent in the current iteration. This means that the agent stays in the same location. The same thing happens in the next and all following iterations. Consequently the agent no longer moves.
This problem was revealed by the test cases generated and was reported on in some test cases as the fault that the plan under test fails to complete.

**Fault 3: The plan under test fails to complete.**

This fault is the completion failure of the plan MoveTo that performs the A* algorithm to look for a path from a starting location to a destination within a search grid. If the plan fails to find a path in a particular situation when the associated system is executed, the plan will terminate the execution of the system, or throw an exception. The occurrence of an exception will lead to a system error that no action is scheduled for the agent in the current iteration and the agent stops moving in the search grid, due to the same reason as discussed in Fault 2. The details and the problem of the plan is presented in the following.

**The Plan**

The plan accepts the following four groups of input variables as parameters:

1. the starting location \((X_s, Y_s)\), which are two belief variables
2. the destination location \((X_d, X_d)\), which are two event variables
3. the search grid identified by \((X_{\text{size}} \times Y_{\text{size}})\), which are two belief variables

4. and a list of obstacles each of which is a location coordinate \((X_o, Y_o)\) within the search grid. Obstacles are records in a belief \(\text{bel}_\text{obstacles}_\text{dat}\), which has two fields \((x_o, y_o)\).

**The Problem**

The problem in this plan is that in some particular situations the plan fails to find a path and consequently throws an exception or terminates the execution of the system. There are four such situations as follows:

1. If the starting location is identical to the destination \((X_s==X_d\) and \(Y_s==Y_d)\), the plan will throw an exception.

2. If the destination is out of the search grid (i.e. \(X_d<=X_{\text{size}}\) and \(Y_d<=Y_{\text{size}}\)), the plan will terminate the execution of the system \(^{14}\).

3. If the destination is in the location of an obstacle (i.e. a record in the belief \(\text{bel}_\text{obstacles}_\text{dat}\) equals to \((X_d, Y_d))\), the plan will terminate the execution of the system.

4. If the starting location is identical to the location of an obstacle (i.e. a record in the belief \(\text{bel}_\text{obstacles}_\text{dat}\) equals to \((X_s, Y_s))\), the plan will terminate the execution of the system.

In addition, there is the fifth situation that if the starting location is out of the search grid, (i.e. \(X_s>X_{\text{size}}\) and \(Y_s>Y_{\text{size}}\)), the plan will find an incorrect path that is out of the search grid. For example, if the search grid is \((10 \times 10)\), the starting point is \((11, 2)\), the destination is \((9, 8)\) and the obstacles are \((2, 2)\) and \((3, 3)\), the plan will find the following path, parts of which is out of the search grid: \((11, 2) -> (10, 2) -> (9, 2) -> (9, 3) -> (9, 4) -> (9, 5) -> (9, 6) -> (9, 7) -> (9, 8)\).

The five situations above are all conceptually invalid for the plan in the system and ideally should not occur if the plan is activated in a valid situation when the system that contains the plan is executed. “Valid” means the starting position is different from the destination, Neither of them is in the location of an obstacle and both of them are within the search grid. However, when the system is executed, problems in another plan \(\text{MoveToUnexploredArea}\) sometimes do lead to the occurrence of the first (the starting point and the destination is

\(^{14}\)by invoking “\text{System.exit(0)};” in terms of Java
identical) and the third (the destination is an obstacle) invalid situations, causing system errors.

The plan $MoveToUnexploredArea$ decides a destination toward which the agent moves in the search grid, then activates the plan $MoveTo$ as a subtask plan to look for a path to the destination. When the agent is at the location $(0, 0)$ (the top-left position in the search grid, see Figure 6.4), $MoveToUnexploredArea$ incorrectly calculates the destination as $(0, 0)$ and provides it to $MoveTo$\(^{15}\), which is activated with the starting point and the destination both as $(0, 0)$, leading to the occurrence of an exception. As a result the agent stops moving for the rest of the system execution.

\[ \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & (X) \\
0 & & & & & \\
1 & G & & & & \\
2 & & G & & & \\
3 & & & & G & \\
4 & & & & & \\
\end{array} \]

A: agent; G: gold.

\( Figure 6.4: \) Example of Fault 3

The third situation can also occur at the system level. When there is no known gold in the search grid, the plan $MoveToUnexploredArea$ selects a position neighbouring to the agent as the destination and provides it to $MoveTo$. However, $MoveToUnexploredArea$ does not check whether there is an obstacle in the selected position. If this happens, $MoveTo$ terminates the system execution on the basis of the destination being occupied by an obstacle.

We also found that another plan always records the positions of other agents as obstacles before $MoveTo$ is executed. Therefore, if $MoveToUnexploredArea$ selects as a destination a neighbour position that happens to be occupied by another agent, $MoveTo$ will also terminate

\(^{15}\)This occurs only when the agent is located in $(0, 0)$. \textbf{(October 14, 2011)}
The above two situations indicate that system errors may occur if a plan (e.g. *MoveTo*) does not handle invalid inputs properly to avoid unexpected results (e.g. exceptions or system termination) when other plans in the same system do not guarantee that the plan is executed only with valid data.

**Fault Detection**

The test cases generated have evaluated the first three invalid situations mentioned above, as well as the valid situation. Faults of the completion failure of the plan were reported on when the three invalid situations were evaluated in some test cases, due to the occurrence of exceptions or the termination of the system. The fourth and the fifth invalid situations were not evaluated due to incomplete coverage of the test cases on combinations of comparative relationships. The test cases do not provide a comprehensive coverage of combinations of comparative relationships between input variables (e.g. “\(X_s > X_{size}\) and \(Y_s > Y_{size}\)”), although each individual comparative relationship is guaranteed (e.g. “\(X_s > X_{size}\)”). We have considered an improvement for test input generation to comprehensively cover combinations of comparative relationships. The improvement will be discussed later in section 6.6.

The test cases generated in our testing framework evaluate a unit with both valid and invalid data. The evaluation of invalid data for a unit under test can be important, as such invalid data may be provided when the plan is executed in a whole system, due to incorrect belief values or event variable values that have been set up by other plans as was the case in this example. To avoid unexpected execution results a plan should be implemented robustly enough to deal with invalid situations and the test cases generated in our testing framework can detect lack of such robustness.

On the other hand an improvement could be considered to verify if a plan is activated with appropriate event variable values or not. The test descriptor of an event could be extended to allow for the user to specify constraints on event variable values (e.g. valid value ranges). When a plan is tested and an outgoing event is posted, the variable values of the event posted could be checked against the value constraints specified in the event’s test descriptor and a fault could be reported on if mismatches are detected. By doing this, we could verify whether subtask plans that handle that outgoing event are activated with appropriate event variable values as expected.
Fault 4: *The plan under test fails to complete.*

The fault that the plan *MoveToDestination* fails to complete in some test cases is because the plan’s logic implemented is faulty. When the plan accesses the belief *bel_direction_dat* to read a record with particular values, that record does not exist, leading to completion failure of the plan. This will lead to a system error that no action is scheduled for the agent in the current iteration and the agent stops moving in the search grid, due to the same reason as discussed in Fault 2.

**The Plan**

The plan is executed when the agent intends to move from its current position towards a destination in the search grid. The plan decides the moving step of the agent in terms of four directions \(^{16}\) from the current position \((x, y)\) of the agent towards a given destination \((dx, dy)\). The variables \(x\), \(y\), \(dx\) and \(dy\) are carried by the event handled by the plan. The values of \(x\) and \(dx\) are associated with east and west and the values of \(y\) and \(dy\) are associated with north and south. For example, “\(x<dx\) and \(y==dy\)” means \((dx, dy)\) is to the east of \((x, y)\) (see the example in Figure 6.5.).

\[\]

\[A: \text{agent } (x, y); \ G: \text{gold } (dx, dy);.\]

*Figure 6.5: Example of Fault 4*

---

\(^{16}\text{north, south, west and east}\)
The plan accesses the belief `bel_direction_dat`, which stores the moving priorities of four directions, to determine the direction of the moving step. The belief has two fields: `dir` is a key field with four possible values “n”, “s”, “w” and “e”, denoting four directions; `rank` denotes different moving priorities of the four directions and is one of 1, 2, 3, and 4 (smaller numbers indicate higher priorities). The belief should always contain four records respectively with rank=1, 2, 3, 4. Before accessing the belief to determine the moving step, the plan can execute different paths in the code depending on comparisons between the four event variables: x, y, dx and dy. In some execution paths the belief is updated while in some other paths the belief is not. The plan should ensure that the belief keeps 4 records with rank=1, 2, 3, 4 after updating the belief. However, sometimes this does not happen.

**The Problem**

Figure 6.6 shows the code branches implemented by the plan, based on which the plan executes multiple paths. Each path is determined by comparative relationships between four input variables and particular records in the belief. When the plan executes some paths (the paths to Result 3 in Figure 6.6), the belief is not updated properly to guarantee the existence
CHAPTER 6. EVALUATION

```java
public plan MoveToDestination extends Plan {
    line 2: //The belief contains four initial records: (“w”, 1), (“s”, 2), (“n”, 3) and (“e”, 4).
    //The belief has two fields: (dir, rank)
    //dir is a key field that indicates direction and its value is one of “n”, “s”, “w”, “e”
    //rank is an integer value, indicating the moving priority in the associated direction
    #uses data Direction bel\_direction\_dat;
    body() {
        int x, y, dx, dy;
        // four input variables from the triggering event
        line 14: x = emovementinfo\_h.x; y = emovementinfo\_h.y; // x=1, y=1
        line 15: dx = emovementinfo\_h.dx; dy = emovementinfo\_h.dy; //dx=2, dy=1
        ...
        line 30: if(x==dx && y==dy)
            { .../ branch.1: This branch is not executed }
        line 48: else if(x==dx)
            { .../ branch.2: This branch is not executed. }
        line 67: else if(y==dy)
            { if(x < dx)
                { // branch.3: This branch is executed.
                    // The belief bel\_direction\_dat is updated. After being updated,
                    // the belief contains 4 records: (“w”, 4), (“e”, 2), (“s”, 3) and (“n”, 3).
                    ...
                }
            }
    } //The plan fails in line 91 below,
    //because there is no record with rank=1 in bel\_direction\_dat.
    logical String $dir1, $dir2, $dir3, $dir4;
    line 91: bel\_direction\_dat.getDir($dir1,1);
    line 92: bel\_direction\_dat.getDir($dir2,2);
    line 93: bel\_direction\_dat.getDir($dir3,3);
    line 94: bel\_direction\_dat.getDir($dir4,4);
    ...
} // end of body
} // end of plan
```

Figure 6.7: Code of The Plan MoveToDestination

of a record with rank=1 (the direction with the highest moving priority). Consequently, when the plan accesses the belief later to read such a record the plan fails to complete. The following is an example of a system error caused by this problem:

When the agent is at the location (1, 1) and there is a gold at (2, 1) (Figure 6.5), the agent will intend to move to east to pick up the gold, so the plan MoveToDestination will be activated with (x, y)=(1, 1) and (dx, dy)=(2, 1). This matches the condition “x<dx and y==dy”. Since the system sets the initial record in the belief as (“w”, 1), (“s”, 2), (“n”, 3) and (“e”, 4), the plan will execute the branch of path 3 in Figure 6.6. In this path the records in the belief will be updated and become (“w”, 4), (“e”, 2), (“s”, 3) and (“n”, 3),
leaving no record with rank=1 (see the code in line 67, Figure 6.7). When the plan later accesses the belief and tries to read a record with rank=1 (line 91, Figure 6.7), the plan fails to complete. As a result, the agent stops moving and remains stuck for the rest of the game.

**Fault Detection**

The test cases for this plan have detected this problem, which is reported on as a fault of completion failure of the plan in some test cases. However, not all execution paths shown in Figure 6.6 were evaluated by the test cases generated. This is because the test cases do not guarantee a comprehensive coverage of combinations of comparative relationships between input variables: $x$, $y$, $dx$ and $dy$. This limitation has also been presented in Fault 3. An improvement to address this limitation will be discussed later in section 6.6.

The values of $dir$ and $rank$ in the belief records are also important for ensuring coverage of all paths. This has been achieved in our test case generation by declaring these two fields as belief variables (see Figure 6.8), ensuring that 16 ($4 \times 4$) value combinations of these two fields were guaranteed as part of test case inputs. However, this only guarantees the placement of one record in the belief. Therefore an initialisation method was also implemented to ensure that the belief contains four records with rank=1, 2, 3, 4. The method inserted the 3 records according to the value of the record inserted by value assignment. For example, if the record generated using belief variables is (“w”, 1)\textsuperscript{17}, the method will insert three records with respective values of $dir$=“e”, “n”, “s”, and $rank$=2, 3, 4. This reflects the flexibility provided by the use of both declaration of belief input variables and an initialisation method for the initialisation of a belief.

**Fault 5: The plan under test fails to complete.**

This fault is also a plan’s completion failure, due to the reason that in some test cases the plan tried to read a belief record with particular values that did not exist. The fault, unlike the previous examples of a plan’s completion failure, does not appear to cause any error\textsuperscript{17}one of the 16 value combinations
when the associated system is executed. However, we believe that the problem revealed by the fault is a potential risk of the plan and should be reported to the user. The details are discussed in the following.

The plan MoveTo accesses the belief bel\_moveRoute\_dat, which stores each historical step of the agent, to read the information of the agent’s last step, including the direction of the step and the step’s index. The value of the step’s index is bound with a logical variable $iLastUpdate$ (Figure 6.9, line 10). The plan then decides the direction of the current step and stores the information of the current step as a new record into the belief bel\_moveRoute\_dat, with the step’s index incrementally increased (Figure 6.9, line 41).

```
public plan MoveTo extends Plan {
    #uses data Direction bel\_moveRoute\_dat;
    body() {
        ...
        logical int $iLastUpdate;
        // reads a record of the agent’s last step, identified by
        // (iCurrentPosX, iCurrentPosY) which is the agent’s current position
        line 10: if (bel\_moveRoute\_dat.check(iCurrentPosX, iCurrentPosY,
            $LastDir, $iLastUpdate).next()) {
            line 11: // the code executed in condition
            ...
            line 20: } // decides the direction of the next step
            ...
            line 41: bel\_moveRoute\_dat.add(iNextPosX, iNextPosY, NextDir,
                $iLastUpdate.as\_int()+1);
            ...
        } // end of body
    } // end of plan
```

\textit{Figure 6.9: Example Two of Plan Failure}

When the plan inserts a new record into bel\_moveRoute\_dat, the plan assumes that the variable $iLastUpdate$ has been bound with the value of the index of the plan’s last step before reading the variable. However, with the placement of line 41 outside the if clause in line 10-20, it is possible that $iLastUpdate$ is accessed even though a binding was not obtained in line 10 when the if condition was false. This then causes an exception leading to the completion failure of the plan. This problem has been detected as a fault of the plan’s completion failure in some test cases, in which the record of the agent’s previous action does not exist in the belief.

This problem does not cause a system error, because a record of the agent’s last step
has always been inserted into the belief when the plan MoveTo was executed last time to determine the last step 18. In addition, before the plan MoveTo is executed to determine the next step, another plan has checked the existence of such a record and inserted a record if it does not exist. Hence when the plan is executed as part of the system execution, the variable $iLastUpdate can always be bound with a value and the plan never fails.

Nevertheless, there is still a potential risk that the plan MoveTo does not check as a precondition the particular runtime environment required (i.e. a particular record in a belief) by the plan. Hence the correct execution of MoveTo depends on other plans that should set up the runtime environment properly and also depends on previous executions of the plan itself. There is a risk that if the code of these plans are changed for some reason (e.g. bug fixing), the runtime environment may no longer be set up properly (e.g. the particular record required is not inserted). Consequently, runtime errors may occur in MoveTo when a whole system is executed in certain situations. Actually, Fault 3 has shown examples that a plan fails to complete in the system execution because other plans incorrectly set belief values and event variable values accessed by that plan.

In fact, in this case, the system logic always intends the if condition to be true, hence line 41 should be within the if clause thus ensuring that no access to $iLastUpdate could happen if it was not successfully bound. This is a coding error that should be fixed, even though it is currently not causing a system error.

Fault 6: A plan that handles the event under test is never executed.

Multiple faults of this type were detected. In each fault, the context condition of the plan associated with the fault, say $P_1$, is overlapped by the context condition of another plan, say $P_2$, which also handles the same event under test. In some of the faults the context conditions of the two plans are both implemented as true 19. In the other faults the context condition of $P_1$ is a subset of the context condition of $P_2$. Hence when $P_1$ is applicable $P_2$ is also always applicable. Furthermore, in the implementation $P_2$ always has a higher priority for execution than $P_1$ when they are both applicable 20. When the event handled by $P_1$ and $P_2$ was tested, $P_2$ was always selected for execution when both these plans were applicable, so $P_1$ was never executed.

In one such example, the event EAct in an evaluation system is handled by four plans:

---

18 MoveTo is executed everytime when the agent decides the next step to move.
19 implemented as “true;” in JACK
20 Priority in this case is determined by the order of specification.
MoveRandomly, FindGold, PickupGold and MoveToDepot. The event is posted to carry out an action of the agent. The context conditions and the execution priorities of these four plans are presented in Table 6.4. The context condition of MoveRandomly (number of gold ≥ 1) is a subset of that of FindGold (true) and the former’s execution priority is lower than that of the latter. Therefore when MoveRandomly is applicable FindGold is also always applicable and is executed. Furthermore, the event EAct and its handling plans have been implemented in such a way that if FindGold is executed but fails, the agent will no longer select an alternative plan for execution and the goal of EAct will fail. Therefore, MoveRandomly actually will never be executed in any situation when the associated system is executed.

In the investigation of these plans we have found that MoveRandomly seems to be a redundant plan, because the functionality of MoveRandomly has been realised by FindGold. MoveRandomly randomly selects a direction and moves a step in the direction. This plan is implemented as a backup plan and should be executed when the agent cannot decide what exact action to carry out (e.g. go to a gold, pick up a gold or go to the depot). FindGold decides the direction to the nearest gold and moves a step in the direction. If there is no gold in the explored area, the plan will randomly select a direction and moves. Therefore, it is not necessary that MoveRandomly is developed as FindGold has realised its functionality. While the existence of redundant code that is never executed does not impact system behaviour, it certainly impacts understandability and maintenance, so should be detected and addressed.

### 6.6 Improvement of Test Input Generation

This section discusses an improvement in the algorithm of test input generation, to address the issue that combinations of comparison relationships are not considered in generation of test case inputs. The improvement considers combinations of comparative relationships between input variables and values of independent variables. Independent variables are those input variables that do not participate in any comparison statement (i.e. variable $z$ in Fig-

<table>
<thead>
<tr>
<th>execution priority</th>
<th>plan</th>
<th>context condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PickupGold</td>
<td>number of gold &lt;3 and there is a gold in the agent’s current position</td>
</tr>
<tr>
<td>2</td>
<td>MoveToDepot</td>
<td>number of gold ≥ 2</td>
</tr>
<tr>
<td>3</td>
<td>FindGold</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>MoveRandomly</td>
<td>number of gold ≥ 1</td>
</tr>
</tbody>
</table>

*Table 6.4: Context Conditions of Plans Handling EAct*
With the improvement applied, in the generation of test case inputs, two combination sets are respectively generated: a set of value combinations (*value-set* for short) following the current algorithm discussed in section 5.2, and a set of comparison combinations each of which is a combination of comparative relationships and values of independent variables (*comparison-set* for short). Then the value-set is combined with the comparison-set to integrate them to a final set of test cases. There are four main steps of test input generation after the improvement is applied (using the example in Figure 6.10):

![Table 6.5: Value Set of The Example in Figure 6.10](image)

- **Step 1**: The value-set of input variables is generated (refer to section 5.2). In the example the *extended level* 21 is applied to generate value combinations (see Table 6.5).

- **Step 2** Comparative relationships are derived from each comparative statement (refer to section 5.3 for derivation details), as shown in Figure 6.11.
1. $x > dx$, which is a valid relationship
2. $x < dx$, which is a valid relationship
3. $x == dx$, which is a valid relationship

1. $y > dy$, which is a valid relationship
2. $y < dy$, which is a valid relationship
3. $y == dy$, which is a valid relationship

Figure 6.11: The Comparative Relationships of The Example in Figure 6.10

<table>
<thead>
<tr>
<th></th>
<th>$x &lt; dx$</th>
<th>$y &lt; dy$</th>
<th>$z == 1$</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$x &lt; dx$</td>
<td>$y &lt; dy$</td>
<td>$z == 2$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$x &lt; dx$</td>
<td>$y &gt; dy$</td>
<td>$z == 1$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$x &lt; dx$</td>
<td>$y &gt; dy$</td>
<td>$z == 2$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$x &lt; dx$</td>
<td>$y == dy$</td>
<td>$z == 1$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$x &lt; dx$</td>
<td>$y == dy$</td>
<td>$z == 2$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$x &gt; dx$</td>
<td>$y &lt; dy$</td>
<td>$z == 1$</td>
<td></td>
</tr>
</tbody>
</table>

|   |   |   |   |   |
|---|---|---|---|
| 17 | $x == dx$ | $y == dy$ | $z == 1$ |
| 18 | $x == dx$ | $y == dy$ | $z == 2$ |
| 19 | $x < dx$ | $y < dy$ | $z == 0$ | invalid |
| 20 | $x < dx$ | $y > dy$ | $z == 0$ |

Table 6.6: Comparison Set of The Example in Figure 6.10

- **Step 3**: A comparison-set is generated, which consists of combinations of the comparative relationships derived and the values of independent variables. Each combination consists of a comparative relationship derived from each comparative statement and a value of each independent variable, as shown in Table 6.6. The user can also choose the thoroughness level of *basic* or *extended*. In this example the extended level is used. A combination in the comparison-set is valid if it is composed of valid comparative relationships and valid values of independent variables, else it is invalid.

- **Step 4**: The value-set and the comparison-set are integrated to create the final set of test cases following the sub-steps:

---

22 the cartesian product of valid values, plus one additional combination for each invalid value and each invalid comparative relationship, refer to section 5.2
For each comparison combination in the comparison-set, the value-set is examined to check if it covers the combination. “Covers” means that in the value-set there is at least one value combination that satisfies the comparative relationships and independent variable values in the comparison combination and has the same validity as the comparison combination.

If the comparison combination is not covered by the value-set, a new value combination will be generated and inserted into the value-set. The values of the newly inserted value combination are randomly generated, while maintaining the constraint that the values must satisfy the associated comparison combination.

In the example the first six combinations in the comparison-set (see Table 6.6) are not covered by the value-set (see Table 6.5) because of “x<dx”, so six new value combinations are inserted into the value-set, as item 48 to item 53 shown in Table 6.7, which is the final set of test case inputs.

<table>
<thead>
<tr>
<th></th>
<th>x==2</th>
<th>y==1</th>
<th>dx==1</th>
<th>dy==1</th>
<th>z==1</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x==2</td>
<td>y==1</td>
<td>dx==1</td>
<td>dy==1</td>
<td>z==2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>x==3</td>
<td>y==1</td>
<td>dx==1</td>
<td>dy==1</td>
<td>z==1</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>x==3</td>
<td>y==2</td>
<td>dx==2</td>
<td>dy==2</td>
<td>z==1</td>
<td>invalid</td>
</tr>
<tr>
<td>32</td>
<td>x==3</td>
<td>y==2</td>
<td>dx==2</td>
<td>dy==2</td>
<td>z==2</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>x==1</td>
<td>y==0</td>
<td>dx==0</td>
<td>dy==0</td>
<td>z==0</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>x==1</td>
<td>y==0</td>
<td>dx==0</td>
<td>dy==0</td>
<td>z==1</td>
<td>invalid</td>
</tr>
<tr>
<td>47</td>
<td>x==3</td>
<td>y==1</td>
<td>dx==0</td>
<td>dy==2</td>
<td>z==1</td>
<td>valid</td>
</tr>
<tr>
<td>48</td>
<td>x==2</td>
<td>y==1</td>
<td>dx==3</td>
<td>dy==2</td>
<td>z==1</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>x==2</td>
<td>y==1</td>
<td>dx==3</td>
<td>dy==2</td>
<td>z==2</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>x==2</td>
<td>y==3</td>
<td>dx==3</td>
<td>dy==2</td>
<td>z==1</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>x==2</td>
<td>y==3</td>
<td>dx==3</td>
<td>dy==2</td>
<td>z==2</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>x==2</td>
<td>y==2</td>
<td>dx==3</td>
<td>dy==2</td>
<td>z==1</td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>x==2</td>
<td>y==2</td>
<td>dx==3</td>
<td>dy==2</td>
<td>z==2</td>
<td></td>
</tr>
</tbody>
</table>

*Table 6.7: Final Value Combination Set of The Example in Figure 6.10*

The new improvement considers combinations of comparisons and variable values more comprehensively than the current algorithm and addresses incomplete coverage on combinations of comparative relationships.
6.7 Further Discussion on Annotation

The quality of annotation can affect the testing results of a system. In general, mistakes or omissions in the annotation of a system can cause too many false positives in the faults detected or inadequate coverage of execution paths. For example, when a system annotated by a student was tested, over 50% of faults detected were false positives due to mistakes in the annotation of the system. After the annotation was improved, removing most of the mistakes, most of these false positives were no longer reported.

Annotation mistakes we have found are summarised in Table 6.8. They are usually due to incorrect specifications of input variable ranges, absence of parts of test input specification and absence of initialisation methods that should be implemented. In the improvement of annotation quality, three suggestions have been summarised as follows:

<table>
<thead>
<tr>
<th>mistake</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value ranges of an input variable specified were not correct.</td>
<td>An input variable was used in a plan as the index of an array, but a value range specified was out of the array size, leading to an exception when the array was accessed.</td>
</tr>
<tr>
<td>absence of input variables</td>
<td>A variable used by a unit (usually a plan) was not specified as an input variable. Hence the variable was not assigned a value and was NULL when it was accessed, leading to an exception.</td>
</tr>
<tr>
<td>A mandatory comparison between input variables was not specified.</td>
<td>Two variables “maxAgents” and “AgentNo” in a plan had the constraint that “AgentNo” ≤ “maxAgents”, which was not specified as a comparative relationship. Consequently in a test case “AgentNo” was larger than “maxAgents” and an exception was thrown.</td>
</tr>
<tr>
<td>absence of init-methods</td>
<td>A complex object variable accessed by the plan under test requires particular values that can only assigned using an init-method. However, the developer did not implemented such an init-method for value assignment. As a result the variable’s value was NULL when the variable was accessed, leading to an exception.</td>
</tr>
<tr>
<td>absence of belief variables</td>
<td>A plan’s logic implemented different execution paths that are determined by different record values of a belief. However, the fields of the belief were not specified as belief variables, so the test cases generated did not cover all execution paths of the plan.</td>
</tr>
</tbody>
</table>

Table 6.8: Examples of Annotation Mistakes

1. First, input variables and comparative relationships should be specified if they affect the execution of the unit under test. For example, variables that participate in the context condition of a plan and variables that determine different execution paths of a
plan should be specified as input variables.

2. Second, the beliefs accessed by the unit usually should be initialised by specifying belief input variables. An initialisation method is implemented for initialising beliefs when particular constraints are required by the unit’s logic, such as multiple particular records of a belief required.

3. Lastly, an initialisation method is usually implemented for:

   (a) setting system properties,
   (b) establishing connections to external programs if necessary,
   (c) instantiating interactee agents that interact with the unit under test,
   (d) initialising those variables that cannot be specified as an input variable, such as a multidimensional array, or an object array in which each element is a complex object variable that requires particular property values,
   (e) performing other operations the developer thinks necessary.

Annotation is a manual operation and requires understanding of the testing framework and the application implemented by the system. Although the quality of annotation still depends on views and experiences of the developer/tester, the quality can be improved if the suggestions above are followed.

Summary

In this chapter details of the evaluation for the testing framework were discussed. Evaluation objectives and the experimental process have been presented, the application for evaluation has been introduced and analysis and conclusion of experimental results have been discussed. In the experiments the actual problems revealed by the faults detected indicate that plans are more problem-prone than events and beliefs (refer to “Incomplete Implementation and Incorrect Implementation” in section 6.4.2), with beliefs least problem-prone (refer to section 6.4.5); specification of context conditions of plans and coverage/overlap of events are often omitted by developers (refer to “Incomplete Design” in section 6.4.2); and the design and the implementation of the system under development may sometimes not be updated synchronously (refer to “Unclassified Mismatches” in section 6.4.2). For the false positives and redundant faults identified, we have discussed possible improvements in different aspects.
of the testing framework to avoid them in the first place, or to provide more precise fault descriptions. The testing process and its relationships to unit dependencies and time efficiency have also been discussed. We have analysed some faults in detail, revealing the underlying problems that lead to these faults and the issues caused by these problems in system execution. Furthermore, a possible improvement on test input generation has been discussed to improve the coverage of the test cases generated using comparative relationships. Finally, the quality of the annotation process for a system has been discussed.
Chapter 7

Conclusions and Future Work

This thesis has presented an approach for automatically unit testing an agent system. The approach extracts information from the design model of a system under test (SUT), then tests each unit to check whether its behaviour is as expected from the design model. We have identified basic types of units to be tested within an agent system and specified a fault model to describe possible types of faults that may exist in units (in chapter 3). A testing tool has been developed that implements a testing process in which all basic units within an agent system can be automatically tested (in chapter 4). An algorithm has been developed for the generation of test case inputs, which provides comprehensive coverage of variable value combinations, with appropriate size of test inputs (in chapter 5). In this chapter, the contributions of our research are summarised, with comparisons to other existing approaches on agent testing. Then limitations of our approach are presented and future work is discussed.

7.1 Contributions

The major contribution of the research is an automated testing framework that allows for completely automated testing of the basic units within an agent system. In the framework all the units of a SUT can be tested in an appropriate order, according to dependencies between units. Each unit is tested against its features specified in its design artifact (descriptor), to reveal possible problems. For each unit under test, a test harness is automatically implemented and executed, which generates and executes test cases to test the associated unit. A test report is finally produced that summarises the testing result for a SUT. Although test cases are generated automatically, Manual test cases may also be added by the tester with particular values to verify certain situations if necessary.
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

In the testing framework, dependencies between units have been taken into account that may affect testing results, to determine an appropriate order in which all the units within an agent are tested. Each unit should be tested before all the other units (dependers) that depend on this unit (a dependee) are tested. If a problem in a dependee unit is revealed by an error or an exception detected, then testing of dependers will be halted to allow the dependee unit to be fixed. With this strategy, each unit can be tested individually without problems caused by its dependee units.

Our system automatically implements a test harness for each individual unit. A test harness generates test case inputs based on the input variables specified in the design descriptor of the unit, then runs test cases to test the associated unit. A test harness contains the unit under test and other relevant units, as well as test driver components that set up the test environment, execute the unit under test, observe runtime behaviour and states of the unit under test and analyse test outcomes. A test harness can also execute certain initialisation routines implemented and specified by the user if necessary before the unit under test is executed.

The test harness for each unit is automatically implemented via a code augmentation process as a part of the completely automated testing process for an agent system. In the code augmentation process the implemented code of the SUT is augmented and components of the test harness are embedded. Test code is inserted into the implementation of the SUT in appropriate places, to track the information required for identifying possible faults. Test harness components and the test code inserted vary depending on the types of units and different information that is needed to identify faults.

Because some plans may need to interact with other agents, the test harness for a plan may contain mock agents that simulate some basic interactions of the external agents which interact with the plan, if such interactions are necessary to the plan’s execution. Alternatively, if the external agents that interact with the plan under test have been tested, the user can also specify to use these rather than mock agents.

Our approach has been evaluated using thirteen systems developed by postgraduate students and shown to be successful in identifying both discrepancies between design and implementation and errors in implemented and running systems.

Compared with other existing work for agent testing, ours is the only approach that has implemented a completely automated testing framework that has thoroughly taken into account automated generation and execution of test cases, as well as automated implementation of components for test execution (i.e. test harnesses implemented by code augmentation).

177 (October 14, 2011)
Some existing approaches extend an automated testing framework for Object Oriented Programming (e.g. JUNIT [Tahchiev et al., 2010]) for automated execution of test cases, such as the SUNIT framework [Tiryaki et al., 2007] and Knublauch’s approach [2002] that are both on top of JUNIT, specifying a set of APIs for developers to manually develop test cases that can be executed to automatically test an agent system. Some other work has implemented test driver components such as tester agents (in terms of part of a test harness in our approach) for automated generation or execution of test cases. Rouff [2002] has developed a test agent that can extract message sequences between agents from a message based model of a SUT, then verify interactions between agents. Caire et al. [2004] have developed a set of APIs based on which developers can develop test cases for verifying agent behaviours, including a tester agent that can automatically execute those manually developed test cases. The JAT testing framework developed by Coelho et.al. [2006] also allows for the user to manually develop test cases for testing an agent’s general features, with a tester agent that can automatically execute these test cases. The eCAT tool developed by Nguyen et al. [2008b; 2010] also contains a test agent that can execute test cases and monitor runtime behaviour of the agent under test. However, none of these test components mentioned above (e.g. tester agents and automated test cases) can completely automatically test units within an agent system, with generation of test cases, test execution, test analysis and generation of testing results (i.e. a test report). Automated implementation of test components (e.g. code augmentation in our approach) is also not taken into consideration in other existing work.

7.2 Additional Contributions

Our approach has also contributed to the area of agent testing by identifying basic units within agents, specifying a fault model that describes fault types and developing an algorithm for automated test input generation.

In contrast to much of the work where agents are usually tested as basic units, we identified plans, events and beliefs as the core units for testing. Testing lower level units within agents reduces the complexity of testing an agent system and reveals possible problems as early as possible. In other existing approaches of testing agent systems, only the SUNIT framework [Tiryaki et al., 2007] and the eCAT tool [Nguyen et al., 2010] test plans within an agent. The SUNIT framework allows for the user to manually develop test cases that verify plan-level structures of an individual agent (in terms of unit dependencies in our research) and actions performed by plans (in terms of outgoing events in our research). The eCAT
tool verifies if plans fulfill their goals as expected, but does not verify pre-conditions (context conditions) of plans and the behaviour of plans (e.g. posting outgoing events). In none of the existing work are testing of events and beliefs explicitly taken into account.

We have specified a fault model that generally describes different types of faults that may exist in basic units of an agent system, in order for test harnesses to automatically detect faults in units. The fault model specifies possible features to be verified for each unit type (plans, events and beliefs) and possible types of faults that may occur relevant to each testable feature. Definitions of testable features and fault types are based on agent-specific features that are described in the design artifacts of associated unit types. When a unit is tested, possible faults will be identified according to fault types specified in the fault model.

There has been some existing work on the definition of fault types that describe potential problems of units within an agent system. The JAT testing framework [Coelho et al., 2006] has included a fault model that describes a set of fault types relevant to general agent features such as faults in message interactions and content and the mismatch of protocols between agents. However, these fault types are only used as a reference for developers to manually develop test cases and are not explicitly linked to automated fault detection. Other work on agent testing only specifies and detects particular types of faults, without a fault model that generally describes fault types of agent systems. Rouff’s approach [2002] verifies communications between agents based on a message model of a SUT. Seo et al. [2004] verified the states and behaviours of agents based on a state model of a SUT. Caire et al. [2004] tested an agent system by verifying agent behaviours based on an extended behavioural model of the system called MAZBD (Multi-Agent Zoomable Behaviour Description) diagram. Zheng and Alagar [2005] developed an ESM (Extended State Machine) model of an agent system based on which states of agents can be verified. The SUNIT framework verifies the plan-level structures of an individual agent and actions performed by plans as mentioned above [Tiryaki et al., 2007]. The eCAT tool focuses on fulfillment or lack of fulfillment of goals achieved by agents and plans [Nguyen et al., 2010].

To comprehensively test a unit in various situations, we have developed an algorithm to generate appropriate test case inputs that provide good coverage of the combinations of input variable values, according to variable value ranges specified by the user, with an ability to limit the size of test inputs. The approaches of Equivalence Class Partitioning and Boundary Value Analysis [Burnstein, 2002, page 67] have been used for choosing sample values of each input variable. The approach of combinatorial design [Cohen et al., 1997] has been used for controlling the size of a set of test cases for a unit to an appropriate size, as the size
of test inputs can quickly explode with the increase of the number of input variables and sample values (e.g. 5 variables, each with 5 sample values, will lead to over 3000 \(5^5=3125\) value combinations as inputs). In addition, the user can also manually add test cases with particular input values to verify certain situations of a unit under test.

There is little existing work in testing of agent systems that has discussed the automated generation of test case inputs. eCAT includes an input generator that generates the inputs of test cases based on the interaction ontologies of agents [Nguyen et al., 2008c]. Interaction ontologies define content semantics of agent interactions, including properties (similar to input variables in our research) and value restrictions (similar to value ranges in our research). The approach does not discuss the control of test input size. Our work on test input generation was however done and published prior to this approach [Zhang et al., 2007]. Other researchers do not explicitly discuss automated generation of test case inputs for agent systems.

7.3 Limitations

The research in this thesis focuses on the stage of unit testing, which however is not sufficient for verifying that the whole system works properly as expected. Testing an agent system should also contain stages of high-level verification such as testing interactions between agents, but these stages are beyond the scope of this thesis.

There is a limitation in the algorithm of test input generation. Although combinations of input variable values and comparisons between variables have been taken into account in test input generation, combinations of comparative relationships (e.g. “\(x>y\)” and “\(y>z\)” are not considered. An improvement of test input generation has been considered to address this limitation (section 6.6), but has not been implemented yet.

Our approach for testing a plan does not involve the understanding of internal logic of the plan, therefore we do not test the coverage of the execution paths and code branches of the plan. The tester may however specify value ranges of input variables as certain values in order to cover a plan’s internal logic thoroughly, but this requires manual effort and understanding of the plan’s logic.

We also do not test semantics of units, such as verifying if a plan achieves its goal successfully. However if the semantics of each unit was part of the structured design documentation, it could readily be added to our approach.

In addition, annotation for a SUT is a manual operation, in which mistakes may easily
be made in the specification of value ranges for input variables and in the implementation of initialisation procedures. The quality of the annotation for a SUT varies depending on the tester’s understanding of both the testing framework and the application of the SUT. We have summarised guidelines for annotation in order for the tester to improve annotation quality (refer to section 6.7).

Although our approach attempts to test a system as thoroughly as possible, allowing large numbers of test cases, it is not a formal verification approach, thus can never guarantee system correctness or that a system is free from errors. However, discussion with industry partners has indicated that formal verification is not likely to be viable for full systems so comprehensive testing as in our approach is still required.

Finally, the design of a unit test harness does not explicitly handle concurrent situations in the system under test, such as a plan under test executing its sub-plans concurrently. Existing approaches for testing concurrent systems [Eytani et al., 2008] may be used to solve this limitation (see page 64).

### 7.4 Future Work

Two improvements could be applied to test input generation. First, the algorithm of test input generation needs to be improved to take into account combinations of comparative relationships as discussed in the limitations above. Second, the format of the specification of input variables has supported variables of base types 1, user defined enumerated types, types of user defined classes and one-dimensional arrays. The format can be extended to allow the specification of more complex variable types, such as multi-dimensional arrays and classes with complex data structures.

Another useful improvement would be to verify whether a plan under test posts an outgoing event with appropriate values. When a plan posts an outgoing event to activate a subtask plan, it is possible that variable values carried by that outgoing event are invalid for the subtask plan, leading to execution errors (refer to an example in section 6.5). However the actual problem is in the plan that generates such invalid values. To check such situations, we could allow the user to specify constraints of variable values of an event (e.g. valid value ranges), in the event’s test descriptor. Alternatively, this could potentially be inferred from the specification of the valid value ranges for event variables in the plan handling the event. During the testing of a plan when an outgoing event is posted, the values of the event could

---

1 integer, long, float, double, boolean and string
then be checked against the constraints specified for the event and possible mismatches could be reported. By doing this, the testing tool could verify whether a plan activates its subtask plans as expected.

Code augmentation could be improved to detect some situations that have been identified as false positives, to increase the effectiveness of fault detection (as discussed in page 150). Code augmentation could also be improved to provide more precise information when plans fail to complete to facilitate fault investigation (as discussed in page 152). Also, additional evaluation should be done to ascertain whether some of the modifications suggested to the definition of fault types are warranted (refer to section 6.4.5).

There is still substantial work needing to be done in testing agent systems, such as verification of agent interactions which can have unexpected results. However, the research in this thesis has provided a strong foundation for such further work, providing some assurance that low level units within agents have been thoroughly checked.
Appendix A

Code of Test Driver Components

A.1 The Test-Driver Plan for Testing an Event

Figure A.1 shows the code of the test-driver plan for testing an event before code augmentation, with a keyword (in bold) that denotes the type name of the event under test. Figure A.2 shows the code after code augmentation (for testing the event “BuyBooks_Ev”).

```
public plan TestDriver_Plan extends Plan {
    ....
    #handles event Activation_Message act_ev;
    #posts event <EVENT_UNDER_TEST_TYPE> trigger;
    context() {
        true;
    }
    body() {
        //set up the test inputs
        setTestInputs(ev.getInputs());

        //run the initialisation procedures in the unit level
        runInitProcedures_unit();

        //posts the event to activate the plan under test
        @subtask(trigger);
    }
    ....
}
```

Figure A.1: Code of Test-Driver Plan for An Event
APPENDIX A. CODE OF TEST DRIVER COMPONENTS

Figure A.2: Code of Test-Driver Plan for An Event after Code Augmentation

A.2 The Test-Driver Plan for Testing a Belief

Figure A.3 shows the code of the test-driver plan for testing an event before code augmentation, with a keyword (in bold) that denotes the type name of the event under test. Figure A.4 shows the code after code augmentation (for testing the belief “Book_DB”).

Figure A.3: Code of Test-Driver Plan for A Belief after Code Augmentation
public plan TestDriver_Plan extends Plan {
    ....
    #handles event Activation_Message act_ev;
    context()
    {
        true;
    }
    body()
    {
        //extract the belief under test,
        Book_DB belief_under_test
        = ev.getBeliefUnderTest();
        ....
    }
    ....
}

Figure A.4: Code of Test-Driver Plan for A Belief after Code Augmentation
Appendix B

Backus-Naur Form of Input Variable Declaration

\[
\text{(Declaration)} ::= \{ \langle \text{EnumTypes} \rangle \} \langle \text{Separator} \rangle \{ \langle \text{InputVariables} \rangle \} \langle \text{Separator} \rangle \\
\{ \langle \text{Comparisons} \rangle \} \langle \text{Separator} \rangle \{ \langle \text{TextualDescription} \rangle \}
\]

\[
\text{(Separator)} ::= ; **;
\]

\[
\text{(EnumTypes)} ::= ( \langle \text{enumtype} \rangle, \langle \text{EnumTypeName} \rangle \langle \text{EnumValues} \rangle \rangle )^+
\]

\[
\text{(EnumTypeName)} ::= \text{a string that denotes an enumerated type name}
\]

\[
\text{(EnumValues)} ::= \langle \text{EnumValue} \rangle ( \langle \text{EnumValue} \rangle \rangle )^*
\]

\[
\text{(EnumValue)} ::= \text{one possible value of the associated enumerated type}
\]

\[
\text{(InputVariables)} ::= \langle \langle \text{NumericVariable} \rangle | \langle \text{StringVariable} \rangle | \langle \text{BooleanVariable} \rangle \\
| \langle \text{EnumVariable} \rangle | \langle \text{BeliefVariable} \rangle | \langle \text{ArrayVariable} \rangle | \langle \text{ObjectVariable} \rangle \rangle ^+
\]

\[
\text{(NumericVariable)} ::= ["\langle \text{Scope} \rangle \langle \text{Name} \rangle \langle \text{VarName} \rangle \langle \text{NumericVarType} \rangle \langle \text{NumericDomainInfo} \rangle \langle \text{Value} \rangle \langle \text{NumericValueRange} \rangle \langle \text{Comparison} \rangle \langle \text{VarValue} \rangle \langle \text{ValueRange} \rangle \langle \text{Operator} \rangle \langle \text{Value} \rangle ]^+
\]

\[
\text{(EnumVarType)} ::= \langle \text{integer} \rangle | \langle \text{long} \rangle | \langle \text{float} \rangle | \langle \text{double} \rangle
\]

\[
\text{(NumericDomainInfo)} ::= \langle \langle \text{NumericVarType} \rangle \langle \text{NumericValueRange} \rangle \langle \text{Comparison} \rangle \langle \text{VarValue} \rangle \langle \text{ValueRange} \rangle \langle \text{Operator} \rangle \langle \text{Value} \rangle \rangle ^*
\]

\[
\text{(NumericValueRange)} ::= \langle \langle \text{NumericCompOperator} \rangle \langle \text{VarValue} \rangle \langle \text{VarValue} \rangle \langle \text{Comparison} \rangle \langle \text{Operator} \rangle \langle \text{Value} \rangle \rangle 
\]
APPENDIX B. BACKUS-NAUR FORM OF INPUT VARIABLE DECLARATION

\( (\text{NumericCompOperator}) ::= \text{">" | "<" | "==" | "!=" | ">=\" | "<=\") \)
\( (\text{VarValue}) ::= \text{a numeric value of the type denoted by } (\text{NumericVarType}) \)

\( (\text{StringVariable}) ::= \text{[" (\text{Scope}) "", (\text{VarName}) ",", "string", ""] (\text{StringDomainInfo}) "] \)
\( (\text{StringDomainInfo}) ::= \text{[" (\text{StringValueRange}) ( ",", (\text{StringValueRange}) ) "] } (\text{\Var}) \)
\( (\text{StringValueRange}) ::= (\text{\NonNumericCompOperator}) \text{"=" (\text{VarValue}) "} \)
\( (\text{\NonNumericCompOperator}) ::= \text{"=" | ",=\") \)
\( (\text{VarValue}) ::= \text{a string value} \)

\( (\text{BooleanVariable}) ::= \text{[" (\text{Scope}) "", (\text{VarName}) ",", "boolean", "] (\text{BoolDomainInfo}) "] \)
\( (\text{BoolDomainInfo}) ::= \text{[" (\text{BoolValueRange}) ( ",", (\text{BoolValueRange}) ) "] } (\text{\Var}) \)
\( (\text{BoolValueRange}) ::= (\text{\NonNumericCompOperator}) (\text{\VarValue}) \)
\( (\text{\VarValue}) ::= \text{"true" | "false"} \)

\( (\text{EnumVariable}) ::= \text{[" (\text{Scope}) "", (\text{VarName}) ",", "\textit{\OneEnumType}", "] (\text{EnumDomainInfo}) "] \)
\( (\text{\OneEnumType}) ::= \text{an enumerated type name specified in } (\text{\Enums}) \)
\( (\text{EnumDomainInfo}) ::= \text{[" (\text{EnumValueRange}) ( ",", (\text{EnumValueRange}) ) "] } (\text{\Var}) \)
\( (\text{EnumValueRange}) ::= (\text{\NonNumericCompOperator}) (\text{\VarValue}) \)
\( (\text{\VarValue}) ::= \text{a value of the associated enumerated type} \)

\( (\text{BeliefVariable}) ::= \text{["belief-var", (\BeliefVarName) ",", (\VarTypeAndDomain) "]} \)
\( (\text{\BeliefVarName}) ::= \text{[" (\BeliefName) ",", (\BeliefVarName) ",", (\\Field) "]} \)
\( (\text{\BeliefName}) ::= \text{the name of a belief that has been specified in the design} \)
\( (\text{\\Field}) ::= \text{the name of the associated belief variable in the implementation} \)
\( (\text{\Field}) ::= \text{the name of a field of the associated belief} \)
\( (\text{\VarTypeAndDomain}) ::= (\text{\VarValue}) (\text{\VarTypeAndDomain}) \)
\( (\text{\VarValue}) ::= \text{"string", (\StringDomainInfo)} \) | (\text{\VarValue}) | (\text{\VarValue}) \)
\( (\text{\VarValue}) ::= \text{"boolean", (\BoolDomainInfo)} \) | (\text{\VarValue}) | (\text{\VarValue}) \)

\( (\text{ArrayVariable}) ::= \text{[" (\text{Scope}) "", (\VarName) ",", "\textit{\Array}", "] (\ArrayDomainInfo) "] \)
\( (\text{\ArrayDomainInfo}) ::= \text{[" (\\ArraySize) ",", (\VarTypeAndDomain) "]} \)
\( (\text{\ArraySize}) ::= \text{an integer number that denotes the size of the array} \)
APPENDIX B. BACKUS-NAUR FORM OF INPUT VARIABLE DECLARATION

\[(ObjectVariable) ::= [" (Scope) "," (VarName) "," (ObjVarType) "," (ObjVarDomainInfo) "]\]

\[(VarObjType) ::= the full name (including the package hierarchy) of an object type\]

\[(ObjVarDomainInfo) ::= (ObjCompareOperator) (ObjVarValue)\]

\[(ObjCompareOperator) ::= "==" | "!="\]

\[(ObjVarValue) ::= an instance of the associated object type, can be null\]

\[(Comparisons) ::= ((CompStatement) )\]

\[(CompStatement) ::= (MandatoryCompare) | (CrossCompare)\]

\[(MandatoryCompare) ::= [compare," (CompExpression) (CompOperator) (CompExpression) "]\]

\[(CompOperator) ::= ">" | "<" | "==" | "!=" | ">=" | "<="\]

\[(CompExpression) ::= a mathematics expression that contains one input variable, such as "minPrice", "maxPrice-5" or "10*minPrice/2"\]

\[(CrossCompare) ::= [compare," (CompExpression) (" (CompExpression) )\]

\[(TextualDescription) ::= a textual description\]
Appendix C

Screenshots of The Testing Tool

- Access testing feature from tools menu of PDT

Figure C.1: Accessing Testing Feature from Tools Menu of PDT
• Select implementation and log directories

Figure C.2: Selection of Implementation and Log Directories
APPENDIX C. SCREENSHOTS OF THE TESTING TOOL

- Testing Window
  1. The input to test a system
  2. Units to be tested (listed in order)
  3. Testing control buttons
  4. Message console
  5. User defined test cases

*Figure C.3: Testing Window*
APPENDIX C. SCREENSHOTS OF THE TESTING TOOL

- Start test

  1. Testing information is displayed in the message console.
  2. Click to open the test report (after a testing process completes)

Figure C.4: Starting Test
Appendix D

Testing Results of The Weather Alert System

The weather alerting system has 3 agent types and 30 units (plans, events and beliefs). 13 faults are detected in total, as shown in Table D.1.

<table>
<thead>
<tr>
<th>fault category</th>
<th>#</th>
<th>fault type from fault model</th>
<th>#</th>
</tr>
</thead>
</table>
| incomplete implementation          | 6  | FT\_CC\_VALID: The context condition value of the plan under test is always evaluated as true when the context condition has been specified in the design.  
reason: The context condition is implemented as “true”.  
The event under test is not implemented yet, so the event is not tested. | 2  |
| incorrect implementation           | 2  | FT\_CC\_INVALID: The context condition value of the plan under test is always evaluated as false when the context condition has been specified in the design.  
reason: The context condition have been specified in the design. However there is an error in the implemented context condition, consequently the context condition value is always evaluated as false. | 2  |
| incomplete design                  | 1  | FT\_E\_OVERLAP\_E: There are multiple plans applicable for the event under test in some test case and the developer does not specify overlap is allowed for the event in the design.  
reason: With particular values of input variables, there are multiple applicable plans. | 1  |
| unclassified mismatch               | 2  | FT\_TRIGGER: The plan is not considered based on its triggering event as specified.           | 2  |
### Table D.1: Categories of Faults Detected and the Associated Fault Types of The Weather Alert System

<table>
<thead>
<tr>
<th>Category</th>
<th>Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>false positives</td>
<td>1</td>
<td><strong>FT_OUTEV_NEVER</strong>: A specified outgoing event is never posted out by the plan under test.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>reason.2:</strong> The event is an action for displaying a warning message. Posting of an action in the system is implemented in a specific way: The action name is implemented as a parameter of an object that displays the message in the graph interface of the system.</td>
</tr>
<tr>
<td>redundant faults</td>
<td>1</td>
<td>The event under test is specified being handled by only one plan in the design. The triggering event of that plan in the implementation is not identical to the event under test. Hence the event is not tested.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>reason:</strong> When that plan is tested, a fault of FT_TRIGGER has been identified.</td>
</tr>
</tbody>
</table>
Appendix E

Test Report of The Weather Alert System

SYSTEM TEST REPORT

System Overview:

Design Specification: weatherAlert_v0.3.pdf

System Implementation: weatherAlert project

# of Agents: 3

Test Overview: (click agent name to review agent test report)

<table>
<thead>
<tr>
<th>#</th>
<th>Agent</th>
<th># of units</th>
<th># of faults</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AirportAgent</td>
<td>24</td>
<td>8</td>
<td>7 warnings, 1 error</td>
</tr>
<tr>
<td>2</td>
<td>GUIAgent</td>
<td>4</td>
<td>3</td>
<td>3 warnings</td>
</tr>
<tr>
<td>3</td>
<td>ForecasterAgent</td>
<td>2</td>
<td>2</td>
<td>2 warnings</td>
</tr>
</tbody>
</table>

Log Directory (link)

Figure E.1: System Level Page of Test Report for WeatherAlert
APPENDIX E. TEST REPORT OF THE WEATHER ALERT SYSTEM

AGENT TEST REPORT

Agent Overview:
Agent under test: : AirPortAgent
H. Units: 24
Log Dir: (link)

Test Overview: (click unit name to review unit test report)

<table>
<thead>
<tr>
<th>#</th>
<th>Unit</th>
<th>Unit Type</th>
<th># of test cases</th>
<th># of faults</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>NewData</td>
<td>event</td>
<td>47</td>
<td>1</td>
<td>(FT_EV_DISPLAY_P) Overlap is not specified. With multiple applicable plans in some test cases: Case 1, Case 2, Case 3, Case 4, Case 5, ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Unit</th>
<th>Unit Type</th>
<th># of test cases</th>
<th># of faults</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>ReportReallyUnexpectedIncreasingWindP</td>
<td>plan</td>
<td>1</td>
<td>1</td>
<td>(FT_OC_INVALID) The CC value is always false when the CC is absent in the design.</td>
</tr>
<tr>
<td>12</td>
<td>ReportWindDiscrepancyP</td>
<td>plan</td>
<td>6</td>
<td>1</td>
<td>(FT_OC_INVALID) The CC value is always false when the CC has been specified in the design.</td>
</tr>
<tr>
<td>13</td>
<td>ReportTrendDiscrepancyP</td>
<td>plan</td>
<td>1</td>
<td>2</td>
<td>(FT_OC_INVALID) The CC value is always true when the CC has been specified in the design. (FT_OOUTY_NEVER) There is no event posted at run-time! Outgoing events specified in design: &quot;NonWarning&quot;</td>
</tr>
</tbody>
</table>

List of other units (N.17)

<table>
<thead>
<tr>
<th>#</th>
<th>Unit</th>
<th>Unit Type</th>
<th># of test cases</th>
<th># of faults</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>subscriptions</td>
<td>belief</td>
<td>0</td>
<td>0</td>
<td>The belief is not tested because there is no action rule specified in the design.</td>
</tr>
<tr>
<td>2</td>
<td>PostWarningP</td>
<td>plan</td>
<td>1</td>
<td>0</td>
<td>The test passes.</td>
</tr>
<tr>
<td>3</td>
<td>NewWarningP</td>
<td>event</td>
<td>1</td>
<td>0</td>
<td>The test passes.</td>
</tr>
<tr>
<td>4</td>
<td>ReportReallyHighWindP</td>
<td>plan</td>
<td>3</td>
<td>0</td>
<td>The test passes.</td>
</tr>
</tbody>
</table>

Figure E.2: Agent Level Page of Test Report for WeatherAlert
APPENDIX E. TEST REPORT OF THE WEATHER ALERT SYSTEM

Figure E.3: Unit Level Page of Test Report for WeatherAlert (Overview and Inputs)
APPENDIX E. TEST REPORT OF THE WEATHER ALERT SYSTEM

PART THREE: TEST SUMMARY

Test of Trigger Event:

PASS: The expected triggering event is actually handled by the plan-under-test.
The distribution of Triggered/Not Triggered:
Triggered: 16
Case.1, Case.2, Case.3, Case.4, Case.5, Case.6, Case.7, Case.8, Case.9, Case.10,
Case.11, Case.12, Case.13, Case.14, Case.15, Case.16

Not Triggered: 0

Test of Context Condition (CC):

The distribution of CC values:
TRUE: 0
Case.2, Case.3, Case.6, Case.7, Case.10, Case.11, Case.14, Case.15

FALSE: 8
Case.1, Case.4, Case.5, Case.8, Case.9, Case.12, Case.13, Case.16

Test of Outgoing Events:

List of outgoing events and the numbers of their posting operations:
NewWarning: 8
Case.2, Case.3, Case.6, Case.7, Case.10, Case.11, Case.14, Case.15

Test of Plan Success:

The result distribution of Plan execution:
The plan succeeds (8):
Case.2, Case.3, Case.6, Case.7, Case.10, Case.11, Case.14, Case.15

The plan fails to complete (0):

Figure E.4: Unit Level Page of Test Report for WeatherAlert (Test Summary)
APPENDIX E. TEST REPORT OF THE WEATHER ALERT SYSTEM

PART FOUR: LOGS OF ALL TEST CASES

Quick links of test cases:
- Case 1
- Case 2
- Case 3
- Case 4
- Case 5
- Case 6
- Case 7
- Case 8
- Case 9
- Case 10
- Case 11
- Case 12
- Case 13
- Case 14
- Case 15
- Case 16

Test Case 1---- (back to: cases list, test summary, top)
Input:

<table>
<thead>
<tr>
<th>#</th>
<th>variable name</th>
<th>type</th>
<th>sample value</th>
<th>is legal</th>
<th>domain</th>
<th>index of Equivalence class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ForecastRain_Latest</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ForecastRain_Nearest</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>ForecastRain_Last</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>ForecastRain_Best</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>subscriptions.user</td>
<td>string</td>
<td>&quot;anonymity&quot;</td>
<td>true</td>
<td>&quot;anonymity&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Actually triggered plan: ReportHighRainChange

The value of Context Condition: false

Test Case 2---- (back to: cases list, test summary, top)
Input:

<table>
<thead>
<tr>
<th>#</th>
<th>variable name</th>
<th>type</th>
<th>sample value</th>
<th>is legal</th>
<th>domain</th>
<th>index of Equivalence class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ForecastRain_Latest</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ForecastRain_Nearest</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>ForecastRain_Last</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>ForecastRain_Best</td>
<td>float</td>
<td>10.1</td>
<td>true</td>
<td>(10.0, +∞)</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>subscriptions.user</td>
<td>string</td>
<td>&quot;anonymity&quot;</td>
<td>true</td>
<td>&quot;anonymity&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Actually triggered plan: ReportHighRainChange

The value of Context Condition: true

List of outgoing events:
- NewWarning

The Plan SUCCEEDS.

Test Case 3---- (back to: cases list, test summary, top)
Input:

<table>
<thead>
<tr>
<th>#</th>
<th>variable name</th>
<th>type</th>
<th>sample value</th>
<th>is legal</th>
<th>domain</th>
<th>index of Equivalence class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ForecastRain_Latest</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ForecastRain_Nearest</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>ForecastRain_Last</td>
<td>float</td>
<td>10.6</td>
<td>true</td>
<td>(10.0, +∞)</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>ForecastRain_Best</td>
<td>float</td>
<td>9.9</td>
<td>true</td>
<td>(−∞, 10.0)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>subscriptions.user</td>
<td>string</td>
<td>&quot;anonymity&quot;</td>
<td>true</td>
<td>&quot;anonymity&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>

Actually triggered plan: ReportHighRainChange

The value of Context Condition: true

List of outgoing events:
- NewWarning

The Plan SUCCEEDS.

... ...

Figure E.5: Unit Level Page of Test Report for WeatherAlert (Details of Cases)
## Appendix F

### Distribution of Faults

<table>
<thead>
<tr>
<th>fault category</th>
<th>#</th>
<th>fault type from fault model</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>incomplete implementation</td>
<td>84</td>
<td>FT_CC_VALID: The context condition value of the plan under test is always evaluated as true when the context condition has been specified in the design.</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reason: The context condition is implemented as “true”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT_CC_INVALID: The context condition value of the plan under test is always evaluated as false when the context condition has been specified in the design.</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reason: The plan is not completely implemented, so its context condition is implemented as “false” to avoid the plan being activated for execution.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT_OUTEV_NEVER: A specified outgoing event is never posted out by the plan under test.</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reason: The code of posting the event is not implemented.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT_CC_INVALID: The context condition value of the plan under test is always evaluated as false when the context condition has been specified in the design.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reason: The context condition is implemented as “false;” in JACK.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT_TRIGGER: The plan is not considered based on its triggering event as specified.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reason: The plan accesses a belief which posts an event that has no handling plan. Therefore the plan is failed to be initialised.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FT_BE_NOT_POST: The belief under test does not post out the event as specified after the operation is successfully applied.</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reason: The code of posting the event is not implemented, although the associated success-triggered and attempt-triggered callback methods have been implemented.</td>
<td></td>
</tr>
</tbody>
</table>
## APPENDIX F. DISTRIBUTION OF FAULTS

<table>
<thead>
<tr>
<th>Category</th>
<th>FT Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>incorrect implementation</td>
<td>FT_EV_NOT_EXEC</td>
<td>A plan that handles the event under test is never executed.</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>When the plan is applicable, there is always another plan with the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>higher execution priority also applicable. Hence the latter plan is executed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and consequently the former one is never executed. see section 6.5 for some</td>
</tr>
<tr>
<td></td>
<td></td>
<td>examples.</td>
</tr>
<tr>
<td>FT_COMPLETION</td>
<td></td>
<td>The plan under test fails to complete in some test cases.</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>They are due to implementation errors in the plan’s body, see section 6.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for some examples.</td>
</tr>
<tr>
<td>incomplete design</td>
<td>FT_CC_ABSENCE</td>
<td>The plan under test is not applicable in some test cases even though the</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>context condition is absent in the design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The context condition is not specified in the design but has been</td>
</tr>
<tr>
<td></td>
<td></td>
<td>implemented.</td>
</tr>
<tr>
<td>FT_OUTEV_NOT</td>
<td></td>
<td>An event posted at runtime by the plan under test is not specified as an</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>outgoing event in the design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The event is not specified as being posted or handled by any plan in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>design.</td>
</tr>
<tr>
<td>FT_EV_INCOMP</td>
<td></td>
<td>There is no plan applicable for the event under test in some test case and</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>the developer does not specify incomplete coverage is allowed for the event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in the design.</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>With some particular values of input variables, there is not applicable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>plan. (# 10)</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>The event is handled by 2 plan in the design, but by 2 additional</td>
</tr>
<tr>
<td></td>
<td></td>
<td>plans in the implementation (by 4 plans in total in the implementation). see</td>
</tr>
<tr>
<td></td>
<td></td>
<td>section 6.5 for details (# 1).</td>
</tr>
<tr>
<td>FT_EV_OVERLAP</td>
<td></td>
<td>There are multiple plans applicable for the event under test in some test</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>case and the developer does not specify overlap is allowed for the event in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With particular values of input variables, there are multiple applicable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>plans.</td>
</tr>
<tr>
<td>unclassified mismatch</td>
<td>FT_TRIGGER</td>
<td>The plan is not considered based on its triggering event as specified.</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>The triggering event in the design and in the implementation is inconsistent.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FT_OUTEV_NEVER</td>
<td>A specified outgoing event is never posted out by the plan under test.</td>
</tr>
<tr>
<td></td>
<td>reason:</td>
<td>The sub-plan that handles the event is inconsistent in the design and in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the implementation. Consequently the test harnesses does not know which</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sub-plan handling the event at runtime and cannot observe the handling of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the event.</td>
</tr>
</tbody>
</table>

(October 14, 2011)
| FT_OUTEV_NOT: | An event posted at runtime by the plan under test is not specified as an outgoing event in the design. |
| reason: | The event is not posted by the plan in the design. |
| FT_COMPLETION: | The plan under test fails to complete in some test cases. |
| reason: | A sub-plan of the plan under test in the implementation is not specified as a sub-plan of the plan under test in the design, but a sub-plan of another plan. Hence the sub-plan is forbidden for execution because it does not belong to the subsystem under test. Consequently the plan under test fails to complete due to the failure of the sub-plan. |
| false positives | 60 |
| (errors) | FT_COMPLETION: The plan under test fails to complete in some test cases. |
| reason: | They are due to application-specific situations. For example, a plan has been designed to terminate the system, so the plan fails to complete due to time-out when it is tested. |
| false positives | 119 |
| (warnings) | FT_CC_VALID: The context condition value of the plan under test is always evaluated as true when the context condition has been specified in the design. |
| reason: | The context condition specified in the design is actually the relevance condition. |
| FT_CC_INVALID: | The context condition value of the plan under test is always evaluated as false when the context condition has been specified in the design. |
| reason: | The context condition is never satisfied in all test cases, as the context condition involves in combinations of comparison relationships between input variables. |
| FT_OUTEV_NEVER: | A specified outgoing event is never posted out by the plan under test. |
| reason.1: | The event posted is handled by a single plan which is in an external library imported thus is not specified in the design. Consequently the test harnesses cannot observe the handling of the event as the plan that handles the event is unknown. (# 26) |
| reason.2: | The event is an action. Posting of an action in the system is implemented in a specific way: The action name is implemented as a parameter of another event posted, thus the test harness did not recognise the posting of the action. (# 2) |

---

1 refer to “Limitation of Execution” in section 4.4 - “Subsystem under Test”
### APPENDIX F. DISTRIBUTION OF FAULTS

<table>
<thead>
<tr>
<th>Fault</th>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FT_OUTEV_NOT</strong></td>
<td>An event posted at runtime by the plan under test is not specified as an outgoing event in the design.</td>
<td>36</td>
</tr>
<tr>
<td>reason:</td>
<td>The event is implemented for debugging purposes, so is not specified.</td>
<td></td>
</tr>
<tr>
<td><strong>FT_CY_EXISTENCE</strong></td>
<td>The cyclic execution does not exist at runtime.</td>
<td>2</td>
</tr>
<tr>
<td>reason:</td>
<td>A cyclic execution should not exist due to the internal logic of the cyclic plan under test.</td>
<td></td>
</tr>
<tr>
<td><strong>FT_EV_INCOMP_W</strong></td>
<td>There is no plan applicable for the event under test in some test case and the developer specifies incomplete coverage is allowed for the event in the design.</td>
<td>19</td>
</tr>
<tr>
<td>reason:</td>
<td>The event under test is handled by only a single plan. In some test cases the context condition of the plan is false, thus no plan is applicable for the event under test.</td>
<td></td>
</tr>
<tr>
<td><strong>FT_EV_OVERLAP_W</strong></td>
<td>There are multiple plans applicable for the event under test in some test case and the developer specifies overlap is allowed for the event in the design.</td>
<td>2</td>
</tr>
<tr>
<td><strong>FT_EV_NOT_EXEC</strong></td>
<td>A plan that handles the event under test is never executed.</td>
<td>3</td>
</tr>
<tr>
<td>reason.1:</td>
<td>The context condition of the plan is never satisfied at runtime, as the context condition involves combinations of comparison relationships between input variables. (# 2)</td>
<td></td>
</tr>
<tr>
<td>reason.2:</td>
<td>The variables in the associated meta-reasoning plan are not considered as input variables of the event under test. (# 1)</td>
<td></td>
</tr>
<tr>
<td><strong>FT_BE_INAPPRO_CB</strong></td>
<td>For an action rule, callback via the implemented attempt-triggered method does not guarantee the success of the operation specified in the rule.</td>
<td>16</td>
</tr>
<tr>
<td>reason:</td>
<td>The fault is detected by code checking. The attempt-triggered method implemented is empty without the code of posting events. Therefore there is no risk that the event is posted by the method when the associated operation fails.</td>
<td></td>
</tr>
<tr>
<td><strong>redundant faults</strong></td>
<td>An exception occurs when the plan under test is executed.</td>
<td>24</td>
</tr>
<tr>
<td>reason:</td>
<td>The same problem has also been identified in “incorrect implementation” category as some faults of type FT_COMPLETION (The plan fails to complete.).</td>
<td>7</td>
</tr>
</tbody>
</table>

(October 14, 2011)
The event is not handled by any plan in the design, thus is not tested.

**reason:** The event is handled by a plan that is not specified in the design because the plan is in an external library. The same problem has also been identified in “false positives (warnings)” as reason.1 of type FT_OUTEV_NEVER (a specified outgoing event not posted).

- **FT_CY_EXISTENCE:** The cyclic execution does not exist at runtime.
  - **reason.1:** The context condition of the cyclic plan is implemented as “false;” in JACK. The same problem has been identified in “incorrect implementation” as a fault of type FT_CC_INVALID (the context condition value always being false). (# 1)
  - **reason.2:** The cyclic plan does not post the event that is part of the cycle, because the code of event posting is not implemented. The same problem has been identified in “incomplete implementation” as a fault of type FT_OUTEV_NEVER (a specified outgoing event not posted). (# 3)

<table>
<thead>
<tr>
<th>Table F.1: Categories of Faults Detected and the Associated Fault Types</th>
</tr>
</thead>
</table>

204 (October 14, 2011)
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


(October 14, 2011)
BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


